

TIBCO SmartSockets™

User's Guide

*Software Release 6.8
July 2006*

Important Information

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE *TIBCO SMARTSOCKETS INSTALLATION GUIDE*). USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document contains confidential information that is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of TIBCO Software Inc.

TIB, TIBCO, Information Bus, The Power of Now, TIBCO Adapter, RTclient, RTserver, RTworks, SmartSockets, and Talarian are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

EJB, J2EE, JMS and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

This software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. Please see the readme.txt file for the availability of this software version on a specific operating system platform.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

Copyright © 1991–2006 TIBCO Software Inc. ALL RIGHTS RESERVED.

TIBCO Software Inc. Confidential Information

Contents

Figures	xvii
Tables	xix
Preface	xxi
Related Documentation	xxii
TIBCO Product Documentation	xxii
Using the Online Documentation	xxii
Conventions Used in This Manual	xxiii
Typeface Conventions	xxiii
Notational Conventions	xxiv
Identifiers	xxiv
Case	xxv
How to Contact TIBCO Support	xxvi
Chapter 1 Messages	1
Message Composition	2
Message Properties	4
Arrival Timestamp	4
Compression	5
Correlation ID	6
Data	7
Delivery Mode	9
Delivery Timeout	11
Destination	12
Expiration	13
Header String Encode	14
Load Balancing Mode	15
Message ID	16
Num Fields	17
Priority	18
Read Only	19
Reference Count	20
Reply To	21
Sender	22
Sender Timestamp	23

Sequence Number	24
Type	25
User-Defined Property	26
Message Types	27
Message Type Properties	28
Compression	28
Delivery Mode	30
Delivery Timeout	31
Grammar	32
Header String Encode	34
Load Balancing Mode	35
Name	36
Number	37
Priority	38
User-Defined Property	39
Standard Message Types	40
User-Defined Message Types	43
Working With Messages	44
Compiling, Linking, and Running	47
Include Files	48
Constructing a Message	49
Accessing the Fields of a Message	52
Accessing Fields by Name	55
Destroying a Message	56
Reusing a Message	57
Advanced Uses of Messages	58
Checking the Type of the Current Field	58
Cloning a Message	59
Array Fields	59
Constructing a Message Within a Message	60
Pointer Fields	61
Unknown Field Values	63
High Performance Guidelines	63
Message Files	64
Text Message Files	64
Binary Message Files	65
Using Message Files	65
Advanced Use of Message Files	66

Chapter 2 Connections	69
Features of Connections	70
Connection Composition	71
Socket	72
Read Buffer	73
Write Buffer	73
Message Queue	74
Block Mode	74
Auto Flush Size	76
Read Timeout	77
Write Timeout	77
Keep Alive Timeout	78
Delivery Timeout	79
GMD Area	80
Thread Synchronization	81
Peer Information	82
Callbacks	84
Sockets	85
Protocols: TCP/IP and Local	85
What is a Socket?	86
How Sockets Work	87
Advantages of Connections Over Sockets	88
Working With Connections	89
Compiling, Linking, and Running	98
Include Files	100
Logical Connection Names	101
Multiple IP Addresses	102
Creating Connections	103
Destroying a Connection	106
Callbacks	107
Receiving and Processing Messages	115
Sending Messages	120
Sending Messages in a Heterogeneous Environment	123
Using Threads With Connections	125
Adding Multiple Threads for a Client	134
Compiling, Linking, and Running	136
Working With Threads and Connections	139

Advanced Uses of Connections	141
Mixing Connections and Xt Intrinsics (Motif)	141
Mixing Connections and the Select Function	143
Mixing Connections and the Windows Message Loop	145
Remote Procedure Calls	147
Time Resolution	147
File Descriptor Upper Limit	148
Handling Network Failures	149
What is Fault Tolerance?	149
Potential Network Failures	150
Keep Alives	151
Blocking and Non-Blocking Read/Write Operations	152
Chapter 3 Publish-Subscribe	153
Publish-Subscribe Overview	154
RTserver and RTclient Composition	156
Projects	156
Subjects	158
What is RTserver?	163
What is RTclient?	166
Ease of Use	169
TIBCO SmartSockets Multicast	170
When Should I Use Multicast?	170
Essential API Functions	172
Message Type Functions:	172
Message Functions:	173
Communication Functions:	174
Utility Functions:	174
Working With RTclient	175
Compiling, Linking, and Running	183
Include Files	186
Differences Between the TipcConn* and TipcSrv* API	186
Setting Options	189
Creating a Connection to RTserver	189
Creating a Connection to RTgms	190
Belonging to a Project	191
Logical Connection Names for RT Processes	192
RTservers with Multiple IP Addresses	197
Finding and Starting RTserver	197
Automatically Reconnecting to RTserver	201
Destroying the Connection to RTserver	202
Using Subjects	203
Callbacks	205

Receiving and Processing Messages	207
Sending Messages	208
Message File Logging	210
Message File Logging Categories	211
Logging Messages	213
Changing Logging Categories	214
Load Balancing	215
Overriding Load Balancing	217
Load Balancing Modes	218
Load Balancing and GMD	219
Load Balancing Example	220
Compiling, Linking, and Running	223
Using Threads with the RTclient API	226
Advanced RTclient Usage	227
Advanced Example With Warm Connections and Server Callbacks	227
Warm Connection to RTserver	235
RTclient-Specific Callbacks	237
Remote Procedure Calls	246
Changing RTclient Options	247
Connecting to Multiple RTservers	248
Using a Dispatcher	249
Types of Dispatchers	249
Single Threaded RTclients	251
Events	254
Multiple Thread Example with Timer and Message Events	264
Message Compression	271
Compressing by Message Type	272
Compressing at the Connection Level	273
Security	275
Basic Security	276
Starting and Stopping RTserver	284
Starting RTserver	288
Stopping RTserver	289
Working with RTserver	290
Setting Options	290
Creating Connections	290
Logical Connection Names	290
Finding Other RTserver Processes	293
Reconnecting to Other RTserver Processes	293
Receiving and Processing Messages from RTclient	293
Message File Logging	295

Dynamic Message Routing	296
Why is Dynamic Message Routing Needed?.	297
Multiple RTserver Processes	298
Distributed Publish-Subscribe Database	300
Lowest Cost Message Routing	301
RTserver Subscribes	303
Network Considerations	305
Controlling Network Bandwidth and Usage	305
Handling Network Failures In Publish Subscribe	307
Chapter 4 Guaranteed Message Delivery	309
Features of GMD.	311
Why is GMD Needed?	312
Loss of Data When Sockets Fail	312
Acknowledgment of Delivery	312
Alternatives to Stream Sockets	313
File-Based and Memory-Based GMD	314
GMD Composition.	315
Sequence Number.	315
GMD Area	316
Delivery Mode	318
GMD Message Types	320
Delivery Timeout	321
Working With GMD	322
Compiling, Linking, and Running	329
Configuring GMD	331
Reverting to Memory-Based GMD.	333
Deleting Files From an Old GMD Area	333
Creating a GMD Area	334
Limiting GMD Resources.	335
Sending Messages	336
Receiving Messages	337
Acknowledging Messages.	338
Waiting for Completion of GMD	338
Resending Messages	339
Receiving Duplicate Messages	340

Handling GMD Failures	341
GMD_FAILURE Messages	341
Delivery Timeout Failures	342
Default Processing of GMD_FAILURE Messages	342
Resending a Message	342
Deleting a Message	343
Limitations of GMD	343
Publish-Subscribe and GMD	344
Delivery Mode in Publish-Subscribe Model	345
Warm RTclient in RTserver	346
GMD Message Types	347
RTclient GMD Considerations	350
Configuring RTclient for GMD	350
DISCONNECT Message Type	351
GMD Area	352
File-based GMD and Connections to Multiple RTservers	353
RTserver GMD Considerations	354
How GMD Works in RTserver	354
Configuring RTserver for GMD	355
Combining GMD and Monitoring	356
Handling GMD Failures with RTclients and RTservers	357
Chapter 5 Project Monitoring	359
Monitoring Overview	360
Monitoring Composition	361
Where Monitoring Information Resides	362
Specifying Items to be Monitored	363
Monitor Scope and T_IPC_MON_ALL	365
Watching or Polling: When to Use	366
Monitoring Message Types	367
Polling	382
Processing Poll Results	386
Polling Message Types	389
Polling Example	400
Compiling, Linking, and Running	405
Watching	408
Processing Watch Results	411
Printing Watch Categories	412
Watching Message Types	413
Watching Example	418
Compiling, Linking, and Running	423

Advanced Monitoring	426
Monitoring With SNMP.	426
Deriving Information.	426
Process Identification.	427
Naming (Directory) Services	428
Running an RTclient With a Hot Backup	429
Compiling, Linking, and Running	439
Chapter 6 Using RTmon.	445
The RTmon Process	446
RTmon Graphical Development Interface.	447
Starting a Graphical Development Interface Session	448
Monitoring Your Project with RTmon GDI	453
Selecting a Project to Monitor	453
Selecting a Command File.	453
Monitoring RTclients.	454
Monitoring Subjects	455
Monitoring Messages Being Received.	456
Monitoring Messages Being Sent	460
Monitoring Server Connections	463
Monitoring RTservers.	464
Sending Messages with RTmon GDI	471
Stopping RTmon GDI Processes	474
RTmon Command Interface	475
Starting a Command Interface Session	475
Chapter 7 Diagnosing Problems.	477
Using RTmon	478
Debugging Messages	478
Debugging Message Types and Message Files	478
Diagnosing Connection Problems.	479
Receiving Unwanted Messages.	479
Diagnosing Memory Problems	480
Diagnosing RTclient Problems	481
Connections and Messages.	481
Why RTclient Is Not Receiving Data	481
Tracing Lost Messages	482
Useful Options	482
Useful Commands	483

Diagnosing RTserver Problems	484
Files Created by RTserver	484
Useful Command-Line Arguments	486
Useful Options	486
Multicast Troubleshooting	487
Verify Your Configuration	487
Verify Your PGM Option Settings	487
Tracing Problems to Their Source	489
Troubleshooting Multicast Problems with Cisco Systems Routers	490
Multicast Testing Tools	491
Summary	491
Chapter 8 Options Reference	493
Setting Option Values	494
RTclient Options	494
RTserver Options	495
RTmon Options	496
Specifying Options	497
Startup Command Files	498
RTclient	498
RTserver	498
RTmon	499
RTclient Options Summary	501
RTserver Options Summary	505
RTmon Options Summary	509
Multi-Thread Mode	512
Option Reference	514
Auth_Data_File	514
Authorize_Publish	514
Backup_Name	515
Catalog_File	515
Catalog_Flags	516
Client_Burst_Interval	516
Client_Connect_Timeout	517
Client_Drain_Subjects	517
Client_Drain_Timeout	518
Client_Keep_Alive_Timeout	519
Client_Max_Buffer	519
Client_Max_Tokens	520
Client_Read_Timeout	520
Client_Reconnect_Timeout	521
Client_Threads	522

Client_Token_Rate	523
Command_Feedback.	523
Compression	524
Compression_Args	524
Compression_Name	525
Compression_Stats	525
Conn_Max_Restarts	526
Conn_Names.	526
Default_Connect_Prefix.	528
Default_Msg_Priority	529
Default_Protocols	529
Default_Subject_Prefix	530
Disable_Mon_Watch_Types	531
Editor	532
Enable_Control_Msgs	533
Enable_Stop_Msgs	534
Gmd_Publish_Timeout	534
Group_Burst_Interval.	535
Group_Max_Buffer.	536
Group_Max_Tokens.	536
Group_Names	537
Group_Token_Rate	538
Ipc_Gmd_Auto_Ack.	538
Ipc_Gmd_Auto_Ack_Policy	539
Ipc_Gmd_Directory	539
Ipc_Gmd_Type	540
Log_In_Client.	540
Log_In_Data	541
Log_In_Group	541
Log_In_Internal	542
Log_In_Msgs	542
Log_In_Server	543
Log_In_Status	543
Log_Out_Client	544
Log_Out_Data	544
Log_Out_Group	544
Log_Out_Internal	545
Log_Out_Msgs	545
Log_Out_Server	545
Log_Out_Status.	546
Max_Client_Conns.	546
Max_Server_Accept_Conns	547
Max_Server_Connect_Conns	547
Max_Server_Conns	548
Monitor_Ident.	549

Monitor_Level	549
Monitor_Scope	550
Multi_Threaded_Mode	551
Project	552
Prompt	552
Proxy_Password	553
Proxy_Username	553
Real_Number_Format	554
Sd_Basic_Acl	554
Sd_Basic_Acl_Timeout	555
Sd_Basic_Admin_Msg_Types	555
Sd_Basic_Trace_File	556
Sd_Basic_Trace_Flags	556
Sd_Basic_Trace_Level	557
Sender_Get_Reply	557
Server_Async_Subscribe	558
Server_Auto_Connect	559
Server_Auto_Flush_Size	559
Server_Burst_Interval	560
Server_Connect_Timeout	560
Server_Connection_Names	561
Server_Delivery_Timeout	562
Server_Disconnect_Mode	563
Server_Gmd_Dir_Name	564
Server_Keep_Alive_Timeout	565
Server_Max_Reconnect_Delay	566
Server_Max_Tokens	566
Server_Msg_Send	567
Server_Names	567
Server_Num_Threads	569
Server_Read_Timeout	571
Server_Reconnect_Interval	572
Server_Start_Delay	572
Server_Start_Max_Tries	573
Server_Start_Timeout	573
Server_Threads	574
Server-Token_Rate	574
Server_Write_Timeout	575
Sm_Security_Driver	575
Socket_Connect_Timeout	576
Srv_Client_Names_Min_Msgs	576
Srv_Subj_Names_Min_Msgs	577
Subjects	577
Time_Format	578
Trace_File	578

Trace_File_Size	579
Trace_Flags	579
Trace_Level	580
Udp_Broadcast_Timeout	580
Unique_Subject	581
Verbose	581
Zero_Recv_Gmd_Failure	582
Chapter 9 Command Reference	583
RTserver Commands	584
Supported RTserver Commands	584
RTclient Commands	585
Supported RTclient Commands	585
RTmon Commands	587
Supported RTmon Commands	587
RTacl Commands	589
Supported RTacl Commands	589
Command Reference	590
alias	591
cd	593
connect	595
create	597
credentials	599
disconnect	600
echo	602
edit	603
evaluate	604
groups	605
help	606
helpopt	607
history	608
load	609
permissions	610
poll	611
quit	615
run	616
send	617
setopt	619

setnpt	620
sh	622
source	623
stats	625
subscribe	626
unalias	628
unsetopt	629
unsubscribe	630
unwatch	631
users	633
watch	634
Chapter 10 Using Multicast	637
Multicast Requirements	639
One-to-Many Communications Solution	640
Features	641
Architecture	642
Multicast Deployment Guidelines	643
RTgms Overview	644
Bandwidth Management	647
Tuning Rate Control	647
Rate Control and Loss	648
Congestion Control	649
RTgms Options	650
RTgms Startup Command Files	652
RTgms Options Summary	653
Option Reference	656
Group_Threshold	656
Setting PGM Options	657
mcast_cm_file	658
PGM Option Summary	658
Pgm_Port	659
Pgm_Receive_Nak_Ttl	660
Pgm_Receive_Pgmcc	661
Pgm_Receive_Pgmcc_Acker_Interval	661
Pgm_Receive_Pgmcc_Loss_Constant	662
Pgm_Source_Admit_High	662
Pgm_Source_Admit_Low	663
Pgm_Source_Group_Ttl	663

- Pgm_Source_Max_Trans_Rate 664
- Pgm_Source_Min_Trans_Rate 665
- Pgm_Source_Pgmcc..... 665
- Pgm_Source_Pgmcc_Acker_Selection_Constant..... 666
- Pgm_Source_Pgmcc_Init_Acker..... 666
- Pgm_Udp_Encapsulation 667
- Starting and Stopping RTgms 668
 - Starting RTgms on UNIX 670
 - Starting RTgms as a Service on Windows..... 670
 - Stopping RTgms 671
- Interrupting RTgms 671
- Sending a Message using Multicast 671
- RTgms Commands..... 672
- Tailoring Your Multicast Deployment..... 673
 - How Multicast Deployment Compares with Unicast Deployment 673
 - Bandwidth Sharing..... 674
 - Client Failovers in Multicast..... 675
 - How Network Devices Forward Multicast 676
 - Multicast and GMD 678
 - UDP Encapsulation of PGM..... 678
 - Multicast Deployment with Frame Relay Networks 679
 - Example Cisco Systems Router Configuration 679
- Index 681**

Figures

Figure 1	Composition of a Typical Message	3
Figure 2	The Flow of Messages Through a Connection.	71
Figure 3	Socket Data Buffering	87
Figure 6	Server Creates a Connection.	103
Figure 7	Client Creates Connection and Rendezvous With Server Connection	103
Figure 8	Server and Client Connection	103
Figure 11	RTserver and RTclient Architecture	157
Figure 12	RTserver Publish-Subscribe Message Routing	164
Figure 13	The Layers of the SmartSockets API	167
Figure 18	Messages Delivered With and Without Load Balancing.	216
Figure 25	Process Connectivity With RTserver Cloud	298
Figure 26	RTserver Groups Connected Using Gateways over a WAN	299
Figure 27	RTserver Cloud with Default Connection Costs	301
Figure 28	RTserver Cloud with Non-Default Connection Costs	302
Figure 29	Benefits of RTserver Subscribes	304
Figure 34	Steps Involved in GMD Successful Delivery	344
Figure 35	RTmon Main Window	448
Figure 36	RTmon Main Window Main Functional Regions	450
Figure 37	Watch Client Time Window	454
Figure 38	Watched Messages Received Window	456
Figure 39	Features of the Watch Messages Received Window.	457
Figure 40	Watch Messages Sent Window	460
Figure 41	Features of the Watch Messages Sent Window	461
Figure 42	Watch Server Connections Graphical Chart	463
Figure 43	Server Information Window	464
Figure 44	Server Information Window (Server Name)	465
Figure 45	RTserver Buffer Windows	467
Figure 46	Send Message Window	471

Figure 47 Multicast Messaging 645

Tables

Table 1	Valid Field Types	7
Table 2	Pseudo Field Types for Message Type Grammar	33
Table 3	Relationship Between Connection Block Mode and Timeout Properties	75
Table 4	Connection Callback Types	114
Table 5	Wildcard Subject Examples	160
Table 6	TipcSrv* Functions With Different Behavior	186
Table 7	TipcConn* Functions Without TipcSrv* Equivalents.	187
Table 8	TipcSrv* Functions Without TipcConn* Equivalents.	188
Table 9	TipcSrv* Functions That do not Automatically Create a Connection to RTserver	200
Table 10	Load Balancing Modes.	218
Table 11	RTclient Callback Types	237
Table 12	Subject Callback Execution	242
Table 13	Message Types that RTserver Processes from RTclient.	293
Table 14	GMD Failure Error Numbers	347
Table 15	RTclient Options.	501
Table 16	RTserver Options	505
Table 17	RTmon Options	509
Table 18	Tuning the Number of I/O Threads	513
Table 19	Server_num_threads	570
Table 20	RTgms Options	653
Table 21	RTclient and RTgms PGM Options	658

Preface

TIBCO SmartSockets is a message-oriented middleware product that enables programs to communicate quickly, reliably, and securely across:

- local area networks (LANs)
- wide area networks (WANs)
- the Internet

TIBCO SmartSockets takes care of network interfaces, guarantees delivery of messages, handles communications protocols, and directs recovery after system or network problems. This enables you to focus on higher-level requirements rather than the underlying complexities of the network.

This reference provides the detailed information you need to understand and use the TIBCO SmartSockets system. Before using this reference, it is helpful to read the *TIBCO SmartSockets Tutorial* and work through the lessons in that book. The *TIBCO SmartSockets Tutorial* begins at a more basic level and is an excellent introduction to SmartSockets.

For an overview of the new features, changes, and enhancements in this Version 6.8 release, see the *TIBCO SmartSockets Installation Guide*.

Topics

- *Related Documentation, page xxii*
- *Conventions Used in This Manual, page xxiii*
- *How to Contact TIBCO Support, page xxvi*

Related Documentation

This section lists documentation resources you may find useful.

TIBCO Product Documentation

The following documents form the TIBCO SmartSockets documentation set:

- *TIBCO SmartSockets API Quick Reference*
- *TIBCO SmartSockets Application Programming Interface*
- *TIBCO SmartSockets C++ User's Guide*
- *TIBCO SmartSockets cxxipc Class Library*
- *TIBCO SmartSockets Installation Guide*
- *TIBCO SmartSockets Java Library User's Guide and Tutorial*
- *TIBCO SmartSockets .NET User's Guide and Tutorial*
- *TIBCO SmartSockets Tutorial*
- *TIBCO SmartSockets User's Guide*
- *TIBCO SmartSockets Utilities*
- *TIBCO SmartSockets C++ and Java Class Libraries*

C++ class library and Java application programming interface (API) materials are available in HTML format only. Access the references through the TIBCO HTML documentation interface.

Using the Online Documentation




The SmartSockets documentation files are available for you to download separately, or you can request a copy of the TIBCO Documentation CD.

Conventions Used in This Manual

This manual uses the following conventions.

Typeface Conventions

This manual uses the following typeface conventions

Example	Use
<code>monospace</code>	This monospace font is used for program output and code example listing and for file names, commands, configuration file parameters, and literal programming elements in running text.
<code>monospace bold</code>	This bold monospace font indicates characters in a command line that you must type exactly as shown. This font is also used for emphasis in code examples.
<i>Italic</i>	<p>Italic text is used as follows:</p> <ul style="list-style-type: none"> • In code examples, file names, etc., for text that should be replaced with an actual value. For example: "Select <i>install-dir</i>/runexample.bat." • For document titles. • For emphasis.
Bold	<p>Bold text indicates actions you take when using a GUI, for example, click OK, or choose Edit from the menu. It is intended to help you skim through procedures when you are familiar with them and just want a reminder.</p> <p>Submenus and options of a menu item are indicated with an angle bracket, for example, Menu > Submenu.</p>
	Warning. The accompanying text describes a condition that severely affects the functioning of the software.
	Note. Be sure you read the accompanying text for important information.
	Tip. The accompanying text may be especially helpful.

Notational Conventions

The notational conventions in the table below are used for describing command syntax. When used in this context, do not type the brackets listed in the table as part of a command line.

Notation	Description	Use
[]	Brackets	Used to enclose an optional item in the command syntax.
< >	Angle Brackets	Used to enclose a name (usually in <i>Italic</i>) that represents an argument for which you substitute a value when you use the command. This convention is not used for XML or HTML examples or other situations where the angle brackets are part of the code.
{ }	Curly Brackets	Used to enclose two or more items among which you can choose only one at a time. Vertical bars () separate the choices within the curly brackets.
...	Ellipsis	Indicates that you can repeat an item any number of times in the command line.

Identifiers

The term identifier is used to refer to a valid character string that names entities created in a SmartSockets application. The string starts with an underscore (`_`) or alphabetic character and is followed by zero or more letters, digits, percent signs (`%`), or underscores. No other special characters are valid. The maximum length of the string is 63 characters. Identifiers are not case-sensitive.

These are examples of valid identifiers:

```
EPS
battery_11
K11
__
_all
```

These are invalid identifiers:

```
20
battery-11
@com
$amount
```


Case

Function names are case-sensitive, and must use the mixed-case format you see in the text. For example, `TipcMsgCreate`, `TipcSrvStop`, and `TipcMonClientMsgTrafficPoll` are SmartSockets functions and must use the case as shown.

Monitoring messages are also case-sensitive, and should be all upper case, such as `T_MT_MON_SERVER_NAMES_POLL_CALL`. This makes it easy to distinguish them from option or function names.

Although option names are not case-sensitive, they are usually presented in text with mixed case, to help distinguish them from commands or other items. For example, `Server_Names`, `Unique_Subject`, and `Project` are all SmartSockets options.

Identifiers used with the products in the SmartSockets family are not case-sensitive. For example, the identifiers `thermal` and `THERMAL` are equivalent in all processes.

In UNIX, shell commands and filenames are case-sensitive, though they might not be in other operating systems, such as Windows. To make it easier to port applications between operating systems, always specify filenames in lower case.

How to Contact TIBCO Support

For comments or problems with this manual or the software it addresses, please contact TIBCO Support as follows.

- For an overview of TIBCO Support, and information about getting started with TIBCO Support, visit this site:

<http://www.tibco.com/services/support>

- If you already have a valid maintenance or support contract, visit this site:

<http://support.tibco.com>

Entry to this site requires a user name and password. If you do not have a user name, you can request one.

Chapter 1 Messages

Within a TIBCO SmartSockets application, interprocess communication occurs through messages. A message is a packet of information sent from one process to one or more other processes providing instructions or data for the receiving process. Messages can carry many different kinds of information, including:

- variable data in the form of a series of variable names and their values, the most common use of a message
- commands to a process's command interface
- user-defined binary data, such as images or multi-byte strings
- monitoring information about RTserver and RTclient processes
- guaranteed message delivery (GMD) information about other messages

This chapter describes message composition, message types, how to create message types, and how to work with messages. Also discussed are message files that are used during the development phase for testing and debugging purposes.

Topics

- *Message Composition, page 2*
- *Message Properties, page 4*
- *Message Types, page 27*
- *Message Type Properties, page 28*
- *Standard Message Types, page 40*
- *User-Defined Message Types, page 43*
- *Working With Messages, page 44*
- *Advanced Uses of Messages, page 58*
- *Message Files, page 64*

Message Composition

All of the different kinds of messages are classified by message types. For example, numeric variable data is typically sent in a `NUMERIC_DATA` type of message, and an operator warning is typically sent in a `WARNING` type of message. A SmartSockets application can use both the standard message types provided with SmartSockets, as well as user-defined message types.

A message (C type `T_IPC_MSG`) is composed of several parts or properties. The most important property is the message data. The other message properties are collectively referred to as the message header.

A message is composed of these properties:

Arrival Timestamp	indicates the time the message was opened for reading.
Compression	is used to enable or disable compression for a message.
Correlation ID	is used for a message ID or an application-specific string.
Data	are the instructions or value part of a message.
Delivery Mode	is the level of guarantee when a message is sent through a connection.
Delivery Timeout	is the number of seconds specifying how long to wait for acknowledgment of delivery of a message sent through a connection.
Destination	is the name of where a message is going.
Expiration	indicates how long, in seconds, the message should be in existence.
Header String Encode	controls whether or not header strings are converted to four-byte integers when a message is sent through a connection.
Load Balancing Mode	is the method of delivery for publish-subscribe operations, which allows a message to be delivered to one or to all subscribing RTclients.
Message ID	uniquely identifies a SmartSockets message.
Num Fields	identifies how many fields are in a particular message.
Priority	is the level of importance of a message.
Read Only	controls whether or not a message can be modified.
Reference Count	is the number of independent references to a message.
Reply To	is the destination or subject where a reply to the messages should be sent.
Sender	is the name of the originator of a message.

Sender Timestamp indicates the time the message was sent.
Sequence Number uniquely identifies a message for guaranteed message delivery.
Type is the kind of message being manipulated.
User-Defined Property is a user-defined value that can be used for any purpose.

Figure 1 shows an example of a standard NUMERIC_DATA message. The data part of this example message is a series of variable name-value pairs (voltage = 33.4534, switch_pos = 0).

Figure 1 Composition of a Typical Message

Message Composition			
Type		NUMERIC_DATA	
Sender		/_workstation1_5415	
Destination		/system/thermal	
Priority		10	
Delivery Mode		T_IPC_DELIVERY_ALL	
Delivery Timeout		20.0	
Load Balancing Mode		T_IPC_LB_WEIGHTED	
Header String Encode		TRUE	
Reference Count		1	
Sequence Number		3892675	
User-Defined Property		42	
Read Only		FALSE	
Data	Field	Type	str
		Value	voltage
	Field	Type	real8
		Value	33.4534
	Field	Type	str
		Value	switch_pos
	Field	Type	real8
		Value	0.0

Message Properties

Arrival Timestamp

Type: Eight-byte real number of C type T_REAL8

Default Value: 0.0

Valid Values: Any valid timestamp

The arrival timestamp property indicates the time a message arrived and was read. As messages are read, their arrival timestamp property is automatically set. A value of 0.0 indicates that the arrival timestamp property was not set.

Function to set value:

TipcMsgSetArrivalTimestamp. For example:

```
if (!TipcMsgSetArrivalTimestamp(msg, TutGetWallTime())) {  
    /* error */  
}
```

Function to get value:

TipcMsgGetArrivalTimestamp. For example:

```
T_REAL8 arrival_timestamp;  
if (!TipcMsgGetArrivalTimestamp(msg, &arrival_timestamp)) {  
    /* error */  
}
```

Compression

Type:	Boolean of C type T_BOOL
Default Value:	The message type compression property is used. The message type compression default is FALSE.
Valid Values:	TRUE or FALSE

The compression property identifies whether compression is enabled. When the compression property is set to TRUE, the fields in the message are automatically compressed when the message is sent. The message is automatically decompressed when a receiver of the message attempts to access any of the message's fields. For a discussion of compression, see Message Compression on page 271.

Function to set value:

TipcMsgSetCompression. For example:

```
if (!TipcMsgSetCompression(msg, T_TRUE)) {  
    /* error */  
}
```

Function to get value:

TipcMsgGetCompression. For example:

```
T_BOOL compress;  
if (!TipcMsgGetCompression(msg, &compress)) {  
    /* error */  
}
```

Correlation ID

Type: Identifier of C type T_STR

Default Value: ""

Valid Values: Any character string

The correlation ID property is typically used for a message ID or for any application-specific string. You can use this property for any purpose.

Function to set value:

TipcMsgSetCorrelationID. For example:

```
T_STR id;
if (!TipcMsgGetMessageId(request_msg, &id)) {
    /* error */
}
if (!TipcMsgSetCorrelationId(reply_msg, id)) {
    /* error */
}
```

Function to get value:

TipcMsgGetMessageId. For example:

```
T_STR id;
if (!TipcMsgGetCorrelationId(msg, &id)) {
    /* error */
}
TutOut("msg had identifier %s\n", id ? id : "<NULL>");
```


Data

Type: String
Default Value: None
Valid Values: Any combination of fields, each containing a type property and a value property

The message data is the property of the message that provides instructions or values for the receiving process. The message data consists of fields that carry a unit of information. Message data can contain any number of fields, although most messages have a well-defined layout for their fields. Each field has the properties type and value.

Type

Type identifies the format of the field value, such as integer, character, or string. The valid field types are:

Table 1 Valid Field Types

Field Type	Meaning
CHAR	One-byte integer
BINARY	Non-restrictive array of characters, such as an entire C data structure or the entire contents of a file
STR	A C string (NULL-terminated array of characters)
STR_ARRAY	Array of STR
INT2	Two-byte integer
INT2_ARRAY	Array of INT2
INT4	Four-byte integer
INT4_ARRAY	Array of INT4
INT8	Eight-byte integer
INT8_ARRAY	Array of INT8
REAL4	Four-byte real number
REAL4_ARRAY	Array of REAL4

Table 1 Valid Field Types

Field Type	Meaning
REAL8	Eight-byte real number
REAL8_ARRAY	Array of REAL8
REAL16	Sixteen-byte real number (not all platforms fully support this type)
REAL16_ARRAY	Array of REAL16
MSG	A message
MSG_ARRAY	Array of MSG
TIMESTAMP	Eight-byte real number representing time as a number of seconds
TIMESTAMP_ARRAY	Array of TIMESTAMP
UTF8	A UTF8 field
UTF8_ARRAY	Array of UTF8
BOOL	A Boolean field
BOOL_ARRAY	Array of BOOL
BYTE	A byte field
XML	XML object

Value

Value identifies the value of the field based on the field type, as well as a value of unknown. See Unknown Field Values on page 63.

Delivery Mode

Type:	Enumerated value of C type <code>T_IPC_DELIVERY_MODE</code>
Default Value:	The message type delivery mode is used. The message type delivery mode default is <code>T_IPC_DELIVERY_BEST_EFFORT</code> .
Valid Values:	<ul style="list-style-type: none"> • <code>T_IPC_DELIVERY_BEST_EFFORT</code> • <code>T_IPC_DELIVERY_ORDERED</code> • <code>T_IPC_DELIVERY_SOME</code> • <code>T_IPC_DELIVERY_ALL</code>

The delivery mode property identifies the level of guarantee when a message is sent through a connection. The possible delivery modes are:

<code>T_IPC_DELIVERY_BEST_EFFORT</code>	No special action, such as acknowledgement, is taken to ensure delivery. If a network or process failure occurs, messages can be lost, and messages might be delivered out of order.
<code>T_IPC_DELIVERY_ORDERED</code>	No special action, such as acknowledgement, is taken to ensure delivery. If a network or process failure occurs, messages can be lost. However, messages are delivered in the order in which they were published.
<code>T_IPC_DELIVERY_SOME</code>	Acknowledgements are used to verify successful delivery. Delivery is treated as successful when at least one receiver acknowledges delivery.
<code>T_IPC_DELIVERY_ALL</code>	Acknowledgements are used to verify successful delivery. Delivery is treated as successful when all receivers acknowledge delivery.

The two modes `T_IPC_DELIVERY_SOME` and `T_IPC_DELIVERY_ALL` are called guaranteed message delivery (GMD), where the sender saves a copy of the message until delivery is successful. For a detailed discussion of GMD, see Chapter 4, Guaranteed Message Delivery.

For applications where the order in which messages are published is critical, or where certain types of clients cannot handle out-of-order messages, use the delivery mode `T_IPC_DELIVERY_ORDERED`. For example, if you have C-based clients sending messages to Java Message Service (JMS) clients, set your delivery mode to `T_IPC_DELIVERY_ORDERED`. GMD delivery modes also ensure that

messages are received in the order in which they were published, even during recovery from network failures. However, GMD uses more system resources, because a copy of every message is kept (in memory or on disk) until the acknowledgements are received. T_IPC_DELIVERY_ORDERED does not guarantee message delivery, but results in faster performance than T_IPC_DELIVERY_SOME and T_IPC_DELIVERY_ALL.

In documentation involving delivery modes, the T_IPC_DELIVERY_ prefix is sometimes dropped outside of code examples (for example, T_IPC_DELIVERY_ALL is discussed as ALL) for the purpose of brevity.

Function to set value:

TipcMsgSetDeliveryMode. For example:

```
if (!TipcMsgSetDeliveryMode(msg, T_IPC_DELIVERY_ALL)) {
    /* error */
}
```

Function to get value:

TipcMsgGetDeliveryMode. For example:

```
if (!TipcMsgGetDeliveryMode(msg, &delivery_mode)) {
    /* error */
}
```

Delivery Timeout

- Type:** Eight-byte real number of C type T_REAL8
- Default Value:** The message type delivery timeout is used. The message type delivery timeout default is unknown (no delivery timeout).
- Valid Values:** Any real number 0 or greater, or unknown

The delivery timeout property identifies how long to wait for guaranteed delivery of a message sent from this process through a connection. The message delivery timeout property is used to override a connection delivery timeout property. To always be used, the message delivery timeout must be greater or equal to the server read timeout, set with the `Server_Read_Timeout` option.

Delivery timeouts are discussed in more detail in Chapter 4, Guaranteed Message Delivery.

Function to set value:

`TipcMsgSetDeliveryTimeout`. For example:

```
if (!TipcMsgSetDeliveryTimeout(msg, 20.0)) {
    /* error */
}
```

Function to get value:

`TipcMsgGetDeliveryTimeout`. For example:

```
if (!TipcMsgGetDeliveryTimeout(msg, &delivery_timeout)) {
    /* error */
}
```

Destination

Type: Identifier of C type T_STR

Default Value: NULL

Valid Values: Any character string

The destination property identifies where a message is going. The meaning of this property depends on how the message is used. When the message is sent using publish-subscribe between RTserver and RTclient, the value is a subject. For a discussion of subjects, see Subjects on page 158. Other applications of messages can use the property for other purposes.

Function to set value:

TipcMsgSetDest. For example:

```
if (!TipcMsgSetDest(msg, "/system/thermal")) {  
    /* error */  
}
```

Function to get value:

TipcMsgGetDest. For example:

```
if (!TipcMsgGetDest(msg, &dest)) {  
    /* error */  
}
```

Expiration

Type: Eight-byte real number of C type T_REAL8

Default Value: 0.0

Valid Values: Any real number 0.0 or greater

The expiration property controls how long, in seconds, the message should be in existence. Applications of messages are free to use the expiration for any purpose. A value of 0.0 indicates that the message does not expire.

Function to set value:

TipcMsgSetExpiration. For example:

```
if (!TipcMsgSetExpiration(msg, 30.0)) {  
    /* error */  
}
```

Function to get value:

TipcMsgGetExpiration. For example:

```
T_REAL8 time_to_live;  
if (!TipcMsgGetExpiration(msg, &time_to_live)) {  
    /* error */  
}
```

Header String Encode

Type:	Boolean of C type T_BOOL
Default Value:	The message type header string encode property is used. The message type header string encode default is FALSE.
Valid Values:	TRUE or FALSE

The header string encode property controls whether the message string properties are converted into four-byte integers when sent through connections. Enabling this property compresses the message header so that less network bandwidth is used. Note that more CPU utilization is required to do the compression.

Function to set value:

TipcMsgSetHeaderStrEncode. For example:

```
if (!TipcMsgSetHeaderStrEncode(msg, TRUE)) {  
    /* error */  
}
```

Function to get value:

TipcMsgGetHeaderStrEncode. For example:

```
if (!TipcMsgGetHeaderStrEncode(msg, &header_str_encode)) {  
    /* error */  
}
```


Load Balancing Mode

Type:	Enumerated value of C type T_IPC_LB_MODE
Default Value:	The message type load balancing mode is used. The message type load balancing mode default is T_IPC_LB_NONE.
Valid Values:	<ul style="list-style-type: none"> • T_IPC_LB_NONE • T_IPC_LB_ROUND_ROBIN • T_IPC_LB_SORTED • T_IPC_LB_WEIGHTED

In normal publish-subscribe operation, a message is sent to all RTclients that have subscribed to the subject the message is being published to. However, in some situations you may wish to have messages sent to only one subscribing RTclient. Load balancing can only be used for RTclient publish-subscribe connections, not peer-to-peer connections. By default, messages are not load balanced and are distributed to all subscribers. For a detailed discussion of load balancing, see Load Balancing on page 215. The four possible load balancing modes are:

T_IPC_LB_NONE	No load balancing (this is the default).
T_IPC_LB_ROUND_ROBIN	Go to each subscriber in turn.
T_IPC_LB_SORTED	Go to the subscriber in lowest lexicographical order (sorted by unique subject).
T_IPC_LB_WEIGHTED	Go to the subscriber with the lightest load (fewest number of unacknowledged messages).

Function to set value:

TipcMsgSetLbMode. For example:

```
if (!TipcMsgSetLbMode(msg, T_IPC_LB_ROUND_ROBIN)) {
    /* error */
}
```

Function to get value:

TipcMsgGetLbMode. For example:

```
if (!TipcMsgGetLbMode(msg, &lb_mode)) {
    /* error */
}
```

Message ID

Type: Identifier of C type T_STR

Default Value: ""

Valid Values: Any character string

The message ID property uniquely identifies a SmartSockets message. No two messages ever have the same message ID. An application can use the message ID property to uniquely identify a message.

Function to set value:

This property is not set, but is generated, using TipcMsgGenerateMessageId. For example:

```
if (!TipcMsgGenerateMessageId(msg)) {
    /* error */
}
```

Function to get value:

TipcMsgGetMessageId. For example:

```
T_STR id;
if (!TipcMsgGetMessageId(msg, &id)) {
    /* error */
}
TutOut("msg had identifier %s\n", id ? id : "<NULL>");
```

Num Fields

Type: Four-byte integer of C type T_INT4

Default Value: 0

Valid Values: Any integer greater than 0

The num fields property identifies how many fields are in a particular message. Assigning this property truncates the message to desired number of fields. A value of 0 empties the data buffer completely.

Function to set value:

TipcMsgSetNumFields. This example empties the data buffer:

```
if (!TipcMsgSetNumFields(msg, 0)) {  
    /* error */  
}
```

Function to get value:

TipcMsgGetNumFields. For example:

```
if (!TipcMsgGetNumFields(msg, &num_fields)) {  
    /* error */  
}
```

Priority

- Type:** Two-byte integer of C type T_INT2
- Default Value:** The message type priority, if set, is used. If the message type priority is unknown, then the message default is Default_Msg_Priority.
- Valid Values:** Any integer from 0 to 65535 inclusive

The priority property identifies the level of importance of a message. The greater the number, the higher the importance. Priority determines the order in which a message is processed after it has been received. It does not affect the order in which messages are published. When a message is sent to a process through a connection, it is placed in the connection message queue of the receiving process in priority order. For a discussion of connection message queues, see Message Queue on page 74.

Function to set value:

TipcMsgSetPriority. For example:

```
if (!TipcMsgSetPriority(msg, 10)) {
    /* error */
}
```

Function to get value:

TipcMsgGetPriority. For example:

```
if (!TipcMsgGetPriority(msg, &priority)) {
    /* error */
}
```

Read Only

Type:	Boolean of C type T_BOOL
Default Value:	<ul style="list-style-type: none">• TRUE if the message is a field within another message• FALSE if the message is separate
Valid Values:	TRUE OR FALSE

The read only property identifies whether or not a message can be modified during processing. Messages included within other messages cannot be modified, while messages that are separate can be modified. For a discussion of messages within messages, see *Constructing a Message Within a Message* on page 60.

Function to set value:

This property is set automatically and cannot be set manually.

Function to get value:

TipcMsgGetReadOnly. For example:

```
if (!TipcMsgGetReadOnly(msg, &read_only)) {  
    /* error */  
}
```

Reference Count

Type: Two-byte integer of C type T_INT2

Default Value: 1

Valid Values: Any integer greater than 0

The reference count property identifies the number of independent references to a message. This property can be used to prevent a message from being destroyed while it is still in use. This is useful when using functions that destroy a message (see Destroying a Message on page 56). Incrementing the reference count of a message is faster and uses less memory than making a complete copy of the message with TipcMsgClone.

Function to set value:

TipcMsgIncrRefCount increments the message reference count. TipcMsgDestroy decrements the message reference count and destroys the message if the reference count drops to zero. For example:

```
if (!TipcMsgIncrRefCount(msg)) {
    /* error */
}
```

Function to get value:

TipcMsgGetRefCount. For example:

```
if (!TipcMsgGetRefCount(msg, &ref_count)) {
    /* error */
}
```

Reply To

Type: Identifier of C type T_STR

Default Value: NULL

Valid Values: Any character string

The reply to property identifies where a reply to a message should be sent. This is used if the reply should be sent to a destination or subject other than the sender. For example, some RT processes, like the gateway, change the sender property when they re-route a message. If you want a reply to be sent to the original sender, you need to set the reply to property to the destination or subject of the sender. For a discussion of subjects, see Subjects on page 158.

Function to set value:

TipcMsgSetReplyTo. For example:

```
if (!TipcMsgSetReplyTo(msg, "reply_subject")) {
    /* error */
}
```

Function to get value:

TipcMsgGetReplyTo. For example:

```
T_STR reply_to_dest;
if (!TipcMsgGetReplyTo(msg, &reply_to_dest)) {
    /* error */
}
```

Sender

Type: Identifier of C type T_STR

Default Value: NULL

Valid Values: Any character string

The sender property identifies the originator of a message. The meaning of this property depends on how the message is used. When the message is sent using publish-subscribe between RTserver and RTclient, the value is the unique subject of the sending process. For a discussion of subjects, see Subjects on page 158. Other applications of messages can use the sender property for other purposes.

Function to set value:

TipcMsgSetSender. For example:

```
if (!TipcMsgSetSender(msg, "_conan_5415")) {  
    /* error */  
}
```

Function to get value:

TipcMsgGetSender. For example:

```
if (!TipcMsgGetSender(msg, &sender)) {  
    /* error */  
}
```


Sender Timestamp

Type: Eight-byte real number of C type T_REAL8

Default Value: 0.0

Valid Values: Any real number 0.0 or greater

The sender timestamp property indicates the time a message was sent. A value of 0.0 indicates that the sender timestamp property was not set.

Function to set value:

TipcMsgSetSenderTimestamp. For example:

```
if (!TipcMsgSetSenderTimestamp(msg, TutGetWallTime())) {
    /* error */
}
```

Function to get value:

TipcMsgGetSenderTimestamp. For example:

```
T_REAL8 sender_timestamp;
if (!TipcMsgGetSenderTimestamp(msg, &sender_timestamp)) {
    /* error */
}
```

Sequence Number

Type: Four-byte integer of C type T_INT4

Default Value: 0

Valid Values: Any integer 0 or greater

The sequence number property uniquely identifies the message for guaranteed message delivery so that duplicate messages can be detected by the receiver(s). When a message with a delivery mode of T_IPC_DELIVERY_SOME or T_IPC_DELIVERY_ALL is sent through a connection, the message sequence number is set to a unique incremented number. The receiving process(es) also records the highest sequence number it has processed. If the message is later resent because of a GMD failure, the receiver can detect the duplicated message by its reused sequence number. A sequence number of zero indicates that a message has not been sent with GMD. For a detailed discussion of GMD, see Chapter 4, Guaranteed Message Delivery.

Function to set value:

This property is set automatically and cannot be set manually.

Function to get value:

TipcMsgGetSeqNum. For example:

```
if (!TipcMsgGetSeqNum(msg, &seq_num)) {
    /* error */
}
TutOut("msg had reply to %s\n", reply_to_dest ? reply_to_dest :
    "<NULL>");
```

Type

Type: Message type data structure of C type T_IPC_MT
Default Value: None. This value must be supplied when creating a message.
Valid Values: Any valid message type

The type property identifies the kind of message being manipulated. For example, a message containing numeric variable values is typically constructed as a NUMERIC_DATA type of message. For a discussion of message types, see Message Types on page 27. For a complete list of standard message types, see Standard Message Types on page 40.

Function to set value:

TipcMsgSetType. For example:

```
mt = TipcMtLookupByNum(T_MT_NUMERIC_DATA);
if (mt == NULL) {
    /* error */
}
if (!TipcMsgSetType(msg, mt)) {
    /* error */
}
```

Function to get value:

TipcMsgGetType. For example:

```
if (!TipcMsgGetType(msg, &mt)) {
    /* error */
}
```

User-Defined Property

- Type:** Four-byte integer of C type T_INT4
- Default Value:** The message type user-defined property is used. The message type user-defined property default is 0.
- Valid Values:** Any integer 0 or greater

The user-defined property can be used for any purpose, such as attaching a version number to messages. This property is not used internally by SmartSockets. When a message is sent through a connection, the user-defined property is sent along with all the other properties.

Function to set value:

TipcMsgSetUserProp. For example:

```
if (!TipcMsgSetUserProp(msg, 42)) {
    /* error */
}
```

Function to get value:

TipcMsgGetUserProp. For example:

```
if (!TipcMsgGetUserProp(msg, &user_prop)) {
    /* error */
}
```

Message Types

Each message has a type property that defines the purpose of the message. A message type can be thought of as a template (such as class) for a specific kind of message, and each message can be considered an instance of a message type. For example, `NUMERIC_DATA` is a message type with a predefined layout requiring a series of name-value pairs, with each string name followed immediately by a numeric value. To send numeric data to a process, the sending process constructs a message that uses the `NUMERIC_DATA` message type. A message type is created once and available for use as the type for any number of messages.

SmartSockets provides a large number of standard message types that you can use and that are also used internally by SmartSockets. When a standard message type does not satisfy a specific need, you can create a user-defined message type. Both standard and user-defined message types are handled in the same manner. Once the message type is created, messages can be constructed, sent, received, and processed through a variety of methods. Standard Message Types on page 40 lists all of the standard message types.

Unlike the properties of a message, which can be changed dynamically, the properties of a message type are only set when the message type is created.

These properties are available for message types:

Compression	identifies the default compression setting for messages of this type.
Delivery Mode	identifies the default delivery mode for messages of this type.
Delivery Timeout	identifies how long to wait for a guaranteed delivery of a message sent from a process through a connection.
Grammar	identifies the layout of fields in messages that use this type.
Header String Encode	controls whether the message string properties are converted into four-byte integers when sent through connections.
Load Balancing Mode	is the method of delivery for publish-subscribe operations, which allows a message to be delivered to one or to all subscribing RTclients.
Name	identifies a unique message type.
Number	identifies the message type by number instead of name.
Priority	identifies the default priority for messages of this type.
User-Defined Property	is a user-defined value that can be used for any purpose.

Message Type Properties

Compression

Type: Boolean of C type T_BOOL

Default Value: FALSE

Valid Values: TRUE or FALSE

The compression property identifies the default compression setting for messages of this type. When the compression property is set to `TRUE`, the fields in the message are automatically compressed when the message is sent. The message is automatically decompressed when a receiver of the message attempts to access any of the message's fields. For a discussion of compression, see [Message Compression](#) on page 271.

The compression setting for an outgoing message can always be set on a per-message basis, but using the message type compression setting makes it easier to change the default setting for all outgoing messages of a specific type.

Function to set value:

`TipcMtSetCompression`. For example:

```
T_IPC_MT mt;
mt = TipcMtLookup("info");
if (mt == NULL) {
    /* error */
}
if (!TipcMtSetCompression(mt, T_TRUE)) {
    /* error */
}
```

Function to get value:

TipcMtGetCompression. For example:

```
T_IPC_MT mt;  
T_BOOL compress;  
mt = TipcMtLookup("info");  
if (mt == NULL) {  
    /* error */  
}  
if (!TipcMtGetCompression(mt, &compress)) {  
    /* error */  
}
```

Delivery Mode

Type:	Enumerated value of C type <code>T_IPC_DELIVERY_MODE</code>
Default Value:	<code>T_IPC_DELIVERY_BEST_EFFORT</code>
Valid Values:	<ul style="list-style-type: none"> • <code>T_IPC_DELIVERY_BEST_EFFORT</code> • <code>T_IPC_DELIVERY_ORDERED</code> • <code>T_IPC_DELIVERY_SOME</code> • <code>T_IPC_DELIVERY_ALL</code>

The delivery mode property identifies the default delivery mode for messages of this type. The message delivery mode property controls the level of guarantee when a message is sent through a connection. When a message is created, its delivery mode is initialized to the message type delivery mode. See [Delivery Mode](#) on page 9.

The delivery mode of an outgoing message can always be set on a per-message basis, but using the message type delivery mode makes it easier to change the default delivery mode for all outgoing messages of a specific type.

Function to set value:

`TipcMtSetDeliveryMode`. For example:

```
if (!TipcMtSetDeliveryMode(mt, T_IPC_DELIVERY_ALL)) {
    /* error */
}
```

Function to get value:

`TipcMtGetDeliveryMode`. For example:

```
if (!TipcMtGetDeliveryMode(mt, &delivery_mode)) {
    /* error */
}
```


Delivery Timeout

Type: Eight-byte real number of C type `T_REAL8`
Default Value: None. This value is unknown when creating a message type.
Valid Values: Any real number greater than 0, or unknown

The delivery timeout property identifies how long to wait for a guaranteed delivery of a message sent from a process through a connection. This timeout is used to check for possible network failures, although at a slightly different level from the read timeout, write timeout, and keep alive timeout (those three are not directly involved with GMD).

The delivery timeout of an outgoing message can always be set on a per-message basis, but using the message type delivery timeout makes it easier to change the default delivery timeout for all outgoing messages of a specific type.

The message type delivery timeout can be set to a specific value or it can be unknown. Note that each message has its own delivery timeout. If the message delivery timeout is not set when the message is sent through a connection, the connection's delivery timeout is used.

If the delivery timeout property of a message is set to zero (0.0), then checking for delivery timeouts is disabled. See Chapter 4, Guaranteed Message Delivery for more information on GMD.

Function to set value:

`TipcMtSetDeliveryTimeout`. For example:

```
if (!TipcMtSetDeliveryTimeout(mt, 20.0)) {
    /* error */
}
```

Function to get value:

`TipcMtGetDeliveryTimeout`. For example:

```
if (!TipcMtGetDeliveryTimeout(mt, &delivery_timeout)) {
    /* error */
}
```

Grammar

Type:	String of C type T_STR
Default Value:	None. This value must be supplied when creating a message type.
Valid Values:	Any valid field type or types (as shown in Table 1 on page 7 and Table 2 on page 33)

The grammar property identifies the layout of fields in messages that use this type. The grammar consists of a list of field types. Each field type in the grammar corresponds to one field in the message. For example, the standard message type TIME has a grammar `real8`, which defines the first and only field as being an eight-byte real number. Standard Message Types on page 40 lists the grammars of all standard message types. Comments (delimited by `/* */` or `(*)`) are also allowed in the grammar.

Function to set value:

This property cannot be changed.

Function to get value:

`TipcMtGetGrammar`. For example:

```
if (!TipcMtGetGrammar(mt, &grammar)) {
    /* error */
}
```

The main purpose of the grammar is to allow messages to be written to text files in a more compact format (see Message Files on page 64 for information on message files). Without a message type grammar, it is difficult to know if the number 45 in a text message file was an INT2, INT4, INT8, REAL4, REAL8, or REAL16 field. Message type grammars also provide some self-documentation for message types. The grammar is not enforced, however, when a message is constructed. A message type with a grammar `STR REAL8`, for example, does not stop a message from being constructed with ten INT4 fields.

Message type grammars do not have any relationship to the data conversion capabilities of connections described in Sending Messages in a Heterogeneous Environment on page 123. SmartSockets messages have strongly-typed fields. The field types, not the grammar, enable connections to perform byte swapping and floating-point conversions.

In addition to the field types shown in Table 1, the field types in Table 2 can be used when defining a message type grammar. They allow values to print in a more readable form when written to a message file.

Table 2 Pseudo Field Types for Message Type Grammar

Field Type	Purpose
ID	Use for the STR field type when the value should be printed without quotes. This is useful for identifiers.
VERBOSE	Use for any field type when both the field type and value should be printed. For example, instead of values printing as <code>voltage 33.4534</code> , they would be printed as <code>str "voltage" real8 33.4534</code> .

There are two common uses for the pseudo field type verbose in message type grammars. The most common usage is when the layout of the fields varies (for example, the type of the second field in the message depends on the value of the first field). The verbose type can also be used when you don't want to use a well-defined grammar. If verbose is used in a message type grammar, it has to be the only field type in the grammar (for example, `"str verbose"` and `"verbose real8 str"` are not valid grammars).

Occasionally, message types use a repetitive group of fields. For example, the NUMERIC_DATA message type allows zero or more name-value pairs. Curly braces ({}) may be used in the message type grammar to indicate such a group. The grammar for the NUMERIC_DATA message type is `"{ id real8 }"` and the grammar for HISTORY_STRING_DATA is `"real8 { id str }"`. Groups must be at the end of the message type grammar, and only one group is allowed for each grammar.

Header String Encode

Type: Boolean of C type T_BOOL

Default Value: FALSE

Valid Values: TRUE or FALSE

The header string encode property controls whether the message string properties are converted into four-byte integers when sent through connections. Enabling this property compresses the message header so that less network bandwidth is used. Note that more CPU utilization is required to do the compression.

The header string encode property of an outgoing message can always be set on a per-message basis, but using the message type header string encode property makes it easier to change the default header string encode property for all outgoing messages of a specific type.

Function to set value:

TipcMtSetHeaderStrEncode. For example:

```
if (!TipcMtSetHeaderStrEncode(mt, TRUE)) {
    /* error */
}
```

Function to get value:

TipcMtGetHeaderStrEncode. For example:

```
if (!TipcMtGetHeaderStrEncode(mt, &header_str_encode)) {
    /* error */
}
```

Load Balancing Mode

Type:	Enumerated value of C type T_IPC_LB_MODE
Default Value:	T_IPC_LB_NONE
Valid Values:	<ul style="list-style-type: none"> • T_IPC_LB_NONE • T_IPC_LB_ROUND_ROBIN • T_IPC_LB_SORTED • T_IPC_LB_WEIGHTED

In normal publish-subscribe operation, a message is sent to all subscribers. However, in some situations you may wish to have messages sent to one subscribing RTclient. Load balancing can only be used for publish-subscribe connections, not peer-to-peer connections. The message load balancing mode overrides the message type load balancing mode. By default, messages are not load balanced and are distributed to all subscribers. For more information, see Load Balancing Mode on page 15 earlier in this chapter. Detailed information is presented in Load Balancing on page 215.

The load balancing mode of an outgoing message can always be set on a per-message basis, but using the message type load balancing mode makes it easier to change the default mode for all outgoing messages of a specific type.

Function to set value:

TipcMtSetLbMode. For example:

```
if (!TipcMtSetLbMode(mt, T_IPC_LB_ROUND_ROBIN)) {
    /* error */
}
```

Function to get value:

TipcMtGetLbMode. For example:

```
if (!TipcMtGetLbMode(mt, &lb_mode)) {
    /* error */
}
```

Name

Type:	Identifier of C type T_STR
Default Value:	None. This value must be supplied when creating a message type.
Valid Values:	Any valid unique message name not starting with an underscore (_)

The name property identifies the message type. Each message type name must be unique. Message type names that start with an underscore are reserved for internal SmartSockets message types.

Function to set value:

This property cannot be changed.

Function to get value:

TipcMtGetName. For example:

```
if (!TipcMtGetName(mt, &name)) {
    /* error */
}
```

Number

- Type:** Four-byte integer of C type T_INT4
- Default Value:** None. This value must be supplied when creating a message type.
- Valid Values:** Any integer greater than 0 that is unique to this message type

The number property also identifies the message type. Each message type number must be unique. The message type number is normally used only when a message is sent between two processes. The message type number is sent instead of the message type name because the message type number almost always uses less storage. The standard SmartSockets message types have message type numbers less than zero. User-defined message types should use numbers greater than zero.

Function to set value:

This property cannot be changed.

Function to get value:

TipcMtGetNum. For example:

```
if (!TipcMtGetNum(mt, &num)) {  
    /* error */  
}
```

Priority

Type: Two-byte integer of C type `T_INT2`
Default Value: None. This value is unknown when creating a message type.
Valid Values: Any integer 0 or greater

The priority property identifies the default priority for messages of this type. The message priority property controls where an incoming message is inserted into a connection's message queue. The message type priority can either be set to a specific value or it can be unknown. When a message is created, its priority is initialized to the message type priority (if set) or to the value of the option `Default_Msg_Priority` (if the message type priority is unknown). Message priorities are discussed in [Priority](#) on page 18.

The priority of an outgoing message can always be set on a per-message basis, but using message type priorities makes it easier to raise or lower the default priority for all outgoing messages of a specific type. Note that if set, the message type priority always overrides the value in the option `Default_Msg_Priority`. The standard SmartSockets message types, by default, all have a priority of unknown, but these can be changed if desired. User-defined message types can use whatever priority you choose.

Function to set value:

`TipcMtSetPriority` and `TipcMtSetPriorityUnknown`. For example:

```
if (!TipcMtSetPriority(mt, 100)) {
    /* error */
}
```

Function to get value:

`TipcMtGetPriority`. For example:

```
if (!TipcMtGetPriority(mt, &priority)) {
    /* error */
}
```


User-Defined Property

Type: Four-byte integer of C type T_INT4

Default Value: 0

Valid Values: Any integer 0 or greater

The user-defined property identifies the default user-defined property for messages of this type. The message user-defined property can be used for any purpose, such as attaching a version number to messages. When a message is created, its user-defined property is initialized to the message type user-defined property. Message user-defined properties are discussed in User-Defined Property on page 26.

The user-defined property of an outgoing message can always be set on a per-message basis, but using the message type user-defined property makes it easier to change the default user-defined property for all outgoing messages of a specific type. The standard SmartSockets message types by default all have a user-defined property of zero (0), but these may be changed if desired. User-defined message types can also use whatever user-defined property is appropriate.

Function to set value:

TipcMtSetUserProp. For example:

```
if (!TipcMtSetUserProp(mt, 42)) {
    /* error */
}
```

Function to get value:

TipcMtGetUserProp. For example:

```
if (!TipcMtGetUserProp(mt, &user_prop)) {
    /* error */
}
```

Standard Message Types

The standard message types are listed in the C header file `msgmt.h`. This file is located in:

UNIX:
`$RTHOME/include/$RTARCH/rtworks`

OpenVMS:
`rthome:[include.rtworks]`

Windows:
`%RTHOME%\include\rtworks`

A standard message type is written differently depending on the context in which it is used. For example, consider the `NUMERIC_DATA` message type. When written to a message file, the message type name is printed as `numeric_data` and is not case sensitive. For a complete discussion of message files, see [Message Files](#) on page 64. When referred to in C code, the message type number is written as `T_MT_NUMERIC_DATA` and is case sensitive.

When messages between RTserver and RTclient are being logged to a message file, the standard message types are divided into three categories: data, status, and internal. For more information on these logging types, see [Message File Logging Categories](#) on page 211.

This table lists all the standard message types. Each grammar element shows the field type followed by a comment that gives a brief description of the field. The monitoring message types (named `MON_*`) are considered standard message types, but are discussed in detail in [Chapter 5, Project Monitoring](#).

Message Type	Grammar	Description
ADMIN_SET	<code>str /*connection or group name*/</code>	Values to dynamically set for options applying to the connection or group channel.
	<code>int4 /*value*/</code>	
	<code>int4 /*value*/</code>	For more information, see the ADMIN_SET message types used for RTserver Options and RTgms Options.
	<code>real8 /*value*/</code>	
BOOLEAN_DATA	<code>{ id /*name*/ bool /*value*/ }</code>	Boolean slot values
CONTROL	<code>str /*command*/</code>	Command for command interface

Message Type	Grammar	Description
ENUM_DATA	{ id /*name*/ id /*value*/ }	Enumerated slot values
GMD_ACK	int4 /*seq_num*/	Internal GMD acknowledgment (used by all connections)
GMD_DELETE	int4 /*seq_num*/	Delete a message in RTserver after a GMD failure (used only from RTclient to RTserver)
GMD_FAILURE	msg /*undelivered_msg*/ str /*failed_process*/ int4 /*err_num*/ real8 /*send_time*/	Unified GMD failure notification built and processed by sender
GMD_INIT_CALL	str /*subject*/	Initialize GMD or load balancing accounting in RTserver for a subject to which messages will be published
GMD_INIT_RESULT	str /*subject*/	GMD or load balancing initialization result from RTserver
GMD_STATUS_CALL	int4 /*seq_num*/	Poll RTserver for GMD status
GMD_STATUS_RESULT	int4 /*seq_num*/ str_array /*success_clients*/ str_array /*failure_clients*/ str_array /*pending_clients*/	GMD status result from RTserver
JMS_BYTES	binary /*byte_array*/	Type of JMS message containing a stream of uninterpreted bytes
JMS_MAP	{str str} /*name_value_pairs*/	Type of JMS message whose body contains a set of name/value pairs where names are strings, and values are primitive types. The entries can be accessed sequentially or randomly by name. The order is undefined.
JMS_OBJECT	binary /*serialized_Java_object*/	Type of JMS message containing a serialized Java object

Message Type	Grammar	Description
JMS_STREAM	verbose <i>/*primitives*/</i>	Type of JMS message containing a stream of primitive values, such as INT or DOUBLE
JMS_TEXT	str <i>/*arbitrary_string*/</i>	Type of JMS message containing a string
NUMERIC_DATA	{ id <i>/*name*/</i> real8 <i>/*value*/</i> }	Numeric slot values
STRING_DATA	{ id <i>/*name*/</i> str <i>/*value*/</i> }	String slot values

For most standard SmartSockets message types, the delivery mode is set to T_IPC_DELIVERY_BEST_EFFORT. Note that for JMS message types, such as JMS_MAP, the delivery mode is set to T_IPC_DELIVERY_ORDERED.

Any message type can be looked up by name with the function TipcMtLookup or looked up by number with the function TipcMtLookupByNum. For example:

```
mt = TipcMtLookup("numeric_data");  
mt = TipcMtLookupByNum(T_MT_STRING_DATA);
```

User-Defined Message Types

When a standard message type does not satisfy a requirement of the application, a new message type can be created. Once created, the user-defined message type is handled in the same manner as a standard message type. To create a user-defined message type, use the function `TipcMtCreate`. This example creates a message type named `XYZ_COORD_DATA`, where messages of type `XYZ_COORD_DATA` have three fields (X, Y, and Z coordinates) that are four-byte integers.

```
#define XYZ_COORD_DATA 1001

mt = TipcMtCreate("xyz_coord_data", XYZ_COORD_DATA, "int4 int4
int4");
if (mt == NULL) {
    /* error */
}
```

If a user-defined message type is going to be used in several programs, it must be created with `TipcMtCreate` in all of those programs. The recommended way of coordinating this in a multi-program project is to place all calls to `TipcMtCreate` in a C/C++ function that all programs then call during initialization. The common function is then placed in a common object library that all programs are linked with. For example of how this can be done, see [Working With RTclient](#) on page 175.



RTserver does not need to call `TipcMtCreate` for user-defined message types to route the user-defined messages.

Working With Messages

This section discusses how to construct, access, and destroy a message. To learn about sending a message, see Chapter 2, Connections. The following example program constructs the NUMERIC_DATA message shown in Figure 1 (except for the sequence number property, which is set automatically when a message is sent through a connection). It also shows how to access the message's values and how to destroy the message. The example code is discussed in detail in the next section.

The source code files for this example are located in these directories:

UNIX:

\$RTHOME/examples/smrtsock/manual

OpenVMS:

RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]

Windows:

%RTHOME%\examples\smrtsock\manual

The online source files have additional `#ifdefs` to provide C++ support. These `#ifdefs` are not shown to simplify the example.

```
/* msg.c -- messages example */
#include <rtworks/ipc.h>

/* ===== */
/*..main -- main program */
int main()
{
    T_IPC_MT mt;
    T_IPC_MSG msg;
    T_STR str_val;
    T_REAL8 real8_val;

    TutOut("Create the message.\n");
    mt = TipcMtLookupByNum(T_MT_NUMERIC_DATA);
    if (mt == NULL) {
        TutOut("Could not look up NUMERIC_DATA msg type: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
}
```

```

msg = TipcMsgCreate(mt);
if (msg == NULL) {
    TutOut("Could not create message: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
TutOut("Set the message properties.\n");
if (!TipcMsgSetSender(msg, "/_conan_5415")) {
    TutOut("Could not set message sender: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgSetDest(msg, "/system/thermal")) {
    TutOut("Could not set message dest: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgSetPriority(msg, 2)) {
    TutOut("Could not set message priority: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgSetDeliveryMode(msg, T_IPC_DELIVERY_ALL)) {
    TutOut("Could not set message delivery mode: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgSetDeliveryTimeout(msg, 20.0)) {
    TutOut("Could not set message delivery timeout: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgSetLbMode(msg, T_IPC_LB_WEIGHTED)) {
    TutOut("Could not set message load balancing mode: error
<%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgSetHeaderStrEncode(msg, TRUE)) {
    TutOut("Could not set message header str encode: error
<%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgSetUserProp(msg, 42)) {
    TutOut("Could not set message user-defined prop: error
<%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
TutOut("Append fields.\n");
if (!TipcMsgAppendStr(msg, "voltage")) {
    TutOut("Could not append first field: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

```

```

if (!TipcMsgAppendReal8(msg, 33.4534)) {
    TutOut("Could not append second field: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgAppendStr(msg, "switch_pos")) {
    TutOut("Could not append third field: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgAppendReal8(msg, 0.0)) {
    TutOut("Could not append fourth field: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
TutOut("Access fields.\n");
if (!TipcMsgSetCurrent(msg, 0)) {
    TutOut("Could not set current field: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgNextStr(msg, &str_val)) {
    TutOut("Could not read first field: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgNextReal8(msg, &real8_val)) {
    TutOut("Could not read second field: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
TutOut("%s = %s\n", str_val, TutRealToStr(real8_val));
if (!TipcMsgNextStr(msg, &str_val)) {
    TutOut("Could not read third field: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgNextReal8(msg, &real8_val)) {
    TutOut("Could not read fourth field: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
TutOut("%s = %s\n", str_val, TutRealToStr(real8_val));
TutOut("Destroy the message.\n");
if (!TipcMsgDestroy(msg)) {
    TutOut("Could not destroy message: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
return T_EXIT_SUCCESS; /* all done */
} /* main */

```


Compiling, Linking, and Running

To compile, link, and run the example program, first you must either copy the program to your own directory or have write permission in these directories:

UNIX:

```
$RTHOME/examples/smrtsock/manual
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]
```

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

Use these commands to compile and link the program:

UNIX:

```
$ rtlink -o msg.x msg.c
```

OpenVMS:

```
$ cc msg.c
$ rtlink /exec=msg.exe msg.obj
```

Windows:

```
$ nmake /f msgw32m.mak
```

On UNIX the `rtlink` command by default uses the `cc` command to compile and link. To use a C++ compiler or a C compiler with a name other than `cc`, set the environment variable `CC` to the name of the compiler, and then `rtlink` uses this compiler. For example, this can be used to compile and link on UNIX with the GNU C++ compiler `g++`:

UNIX:

```
$ env CC=g++ rtlink -o msg.x msg.c
```

Use these commands to run the program:

UNIX:

```
$ msg.x
```

OpenVMS:

```
$ run msg.exe
```

Windows:

```
$ msg.exe
```

The output from the program is:

```
Create the message.
Set the message properties.
Append fields.
Access fields.
voltage = 33.4534
switch_pos = 0
Destroy the message.
```

Include Files

Code written in C or C++ that uses the IPC Application Programming Interface (API) must include the header file `<rtworks/ipc.h>`. This file is located in these directories:

UNIX:

```
$RTHOME/include/$RTARCH/rtworks
```

OpenVMS:

```
RTHOME: [ INCLUDE.RTWORKS ]
```

Windows:

```
%RTHOME%\include\rtworks
```

The SmartSockets IPC API includes all the functions used for interprocess communication.

Constructing a Message

These three steps are required when constructing a message:

1. Create a message of a particular type.
2. Set the header properties of the message.
3. Append fields to the message data.

Step 1 Create a message

A message is created using the `TipcMsgCreate` function with the message type. This returns a message that is filled in later. For example:

```
mt = TipcMtLookupByNum(T_MT_NUMERIC_DATA);
if (mt == NULL) {
    TutOut("Could not look up NUMERIC_DATA msg type: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
msg = TipcMsgCreate(mt);
if (msg == NULL) {
    TutOut("Could not create message: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

Step 2 Set header properties of a message

The header (non-data) properties are set using the `TipcMsg SetProperty` function where *Property* is replaced by the property being set, such as sender or priority. For example:

```
if (!TipcMsgSetSender(msg, "/_conan_5415")) {
    TutOut("Could not set message sender: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgSetDest(msg, "/system/thermal")) {
    TutOut("Could not set message dest: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgSetPriority(msg, 2)) {
    TutOut("Could not set message priority: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgSetDeliveryMode(msg, T_IPC_DELIVERY_ALL)) {
    TutOut("Could not set message delivery mode: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

```

if (!TipcMsgSetDeliveryTimeout(msg, 20.0)) {
    TutOut("Could not set message delivery timeout: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgSetLbMode(msg, T_IPC_LB_WEIGHTED)) {
    TutOut("Could not set message load balancing mode: error
<%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgSetHeaderStrEncode(msg, TRUE)) {
    TutOut("Could not set message header str encode: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgSetUserProp(msg, 42)) {
    TutOut("Could not set message user-defined prop: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

```

Step 3 **Append fields to message data**

Fields are appended to the message using one of a number of append functions. These begin with `TipcMsgAppendType` where *Type* is replaced by the field type, such as `TipcMsgAppendStr`. For example:

```

if (!TipcMsgAppendStr(msg, "voltage")) {
    TutOut("Could not append first field: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgAppendReal8(msg, 33.4534)) {
    TutOut("Could not append second field: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgAppendStr(msg, "switch_pos")) {
    TutOut("Could not append third field: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgAppendReal8(msg, 0.0)) {
    TutOut("Could not append fourth field: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

```

Fields are always appended to the end of the message.

Using the TipcMsgWrite Convenience Function

The TipcMsgWrite convenience function handles a variable number of arguments and allows you to append one or more fields to a message. The function takes enumerated values that begin with `T_IPC_FT_TYPE`, where *TYPE* is replaced by a field type as shown in Table 1, such as `T_IPC_FT_STR`. A final parameter of `NULL` is used to terminate the variable number of arguments. Using this function, the above values could be added to the message.

For example:

```
if (!TipcMsgWrite(msg,
                  T_IPC_FT_STR, "voltage",
                  T_IPC_FT_REAL8, 33.4534,
                  T_IPC_FT_STR, "switch_pos",
                  T_IPC_FT_REAL8, 0.0,
                  NULL)) {
    TutOut("Could not append all fields: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```



C/C++ functions that use a variable number of arguments often require fewer lines of code to use, but there is no type checking done by the C/C++ compiler on the variable arguments.

As an added convenience, TipcMsgWrite also allows enumerated values that begin with `T_IPC_PROP_NAME`, where *NAME* is replaced by a message property name, such as `T_IPC_PROP_DELIVERY_MODE`. For example, the entire message shown above could have been constructed with one call to TipcMsgWrite:

```
if (!TipcMsgWrite(msg,
                  T_IPC_PROP_TYPE, TipcMtLookup("numeric_data"),
                  T_IPC_PROP_SENDER, "/_conan_5415",
                  T_IPC_PROP_DEST, "/system/thermal",
                  T_IPC_PROP_PRIORITY, 2,
                  T_IPC_PROP_DELIVERY_MODE, T_IPC_DELIVERY_ALL,
                  T_IPC_PROP_DELIVERY_TIMEOUT, 20.0,
                  T_IPC_PROP_LB_MODE, T_IPC_LB_WEIGHTED,
                  T_IPC_PROP_HEADER_STR_ENCODE, TRUE,
                  T_IPC_PROP_USER_PROP, 42,
                  T_IPC_FT_STR, "voltage",
                  T_IPC_FT_REAL8, 33.4534,
                  T_IPC_FT_STR, "switch_pos",
                  T_IPC_FT_REAL8, 0.0,
                  NULL)) {
    TutOut("Could not construct message: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

Adding Fields by Name

As an alternative to appending fields to a message, you can also add fields to a message and give each field a name. This allows you to access the field using that name. To add fields by name, use the `TipcMsgAddNamed*` functions, such as `TipcMsgAddNamedStr`. A single message may contain both fields with names and fields without names. There is no conflict.

For example, the following lines of code operate on the same message. One appends an `INT4` and the next adds a string by name, in this case, `string one`:

```
/* Add a non-named int4 field, and a named string field */
TipcMsgAppendInt4(msg, 5);
TipcMsgAddNamedStr(msg, "string one", "hello");
```

A named field is like any other field in the message, except that it also has a name.

Accessing the Fields of a Message

There are two steps involved in accessing the fields of a message. First, the current field in the message must be set, and then it can be accessed.

Setting the Current Field

By default the fields of a message are accessed in the order they are appended. The first field accessed in the previous example is `STR` with the value of `voltage`, the second field accessed is `REAL8` with a value of `33.4534`, and so on.

The field being accessed is considered the current field. When a field has been accessed, the next field becomes the current field. The order in which the fields are accessed is changed using the `TipcMsgSetCurrent` function. For example:

```
if (!TipcMsgSetCurrent(msg, 2)) {
    /* error */
}
```



The first field in a message is considered field 0.

With the current field set to two (2), the field accessed in the previous example would be `str` with a value of `switch_pos`, the next accessed value would be `REAL8` with a value of zero (0.0). Because this is the last field in the message, unless the current field is reset, no other fields would be accessed. Changing the current field does not change the order of the fields or the number of fields in the message.

The number of fields in a message is found using the `TipcMsgGetNumFields` function. For example:

```
if (!TipcMsgGetNumFields(msg, &num_fields)) {
    /* error */
}
```

When beginning to access the fields of a message, the current field should always be set first so that the desired field is the current field. In the above example, the current field is set to the first field as follows:

```
if (!TipcMsgSetCurrent(msg, 0)) {
    /* error */
}
```

Accessing the Current Field

The fields of a message are accessed by making calls to the `TipcMsgNextType` function where *Type* is replaced by the field type, such as `TipcMsgNextStr`. To access the first two fields of the message in Figure 1, a call is made to `TipcMsgNextStr` and `TipcMsgNextReal8`. Something could be done with these values, such as printing them with `TutOut`, and then the next two fields could be accessed. For example:

```
if (!TipcMsgNextStr(msg, &str_val)) {
    TutOut("Could not read first field: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgNextReal8(msg, &real8_val)) {
    TutOut("Could not read second field: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
TutOut("%s = %s\n", str_val, TutRealToStr(real8_val));
if (!TipcMsgNextStr(msg, &str_val)) {
    TutOut("Could not read third field: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgNextReal8(msg, &real8_val)) {
    TutOut("Could not read fourth field: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
TutOut("%s = %s\n", str_val, TutRealToStr(real8_val));
```

All pointer-sized field information, such as `BINARY`, `STR`, `INT4_ARRAY`, and `MSG` values, is accessed with pointers directly into the message data so that no potentially-large memory copies are needed. The data stored in these pointers should not be modified or deallocated (`TipcMsgDestroy` automatically deallocates the entire message data). All non-pointer field information, such as `INT4` and the size of a `REAL8_ARRAY`, is copied into the appropriate parameter to `TipcMsgNextType`.

Using the `TipcMsgRead` Convenience Function

The `TipcMsgRead` convenience function handles a variable number of arguments and allows you to access one or more fields in a message (and also advance the current field). The function takes enumerated values that begin with `T_IPC_FT_`*TYPE*, where *TYPE* is replaced by a field type as shown in Table 1 on page 7, such as `T_IPC_FT_STR`. A final parameter of `NULL` is used to terminate the variable number of arguments. Using this function, the first two values could be accessed as follows:

```
if (!TipcMsgRead(msg,
                 T_IPC_FT_STR, &str_val,
                 T_IPC_FT_REAL8, &real8_val,
                 NULL)) {
    TutOut("Could not read first two fields: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
TutOut("%s = %s\n", str_val, TutRealToStr(real8_val));
```

As an added convenience, `TipcMsgRead` also allows enumerated values that begin with `T_IPC_PROP_NAME`, where *NAME* is replaced by a message property name, such as `T_IPC_PROP_DELIVERY_MODE`.

Accessing Fields by Name

If a message has fields that were added by name, you can access those fields using that name. This is a different approach from the sequential access described in the previous sections. A field that has a name may be accessed either by that name or sequentially.

This example shows accessing fields from a message that contains both named and unnamed fields. It also shows how to get the name of the current field:

```
#include <rtworks/ipc.h>

/*=====*/
/*..main -- named fields example */
void main(int argc, char **argv)
{
    T_IPC_MSG msg;
    T_STR str;
    T_INT4 i;

    /* Create the message */
    msg = TipcMsgCreate(TipcMtLookupByNum(T_MT_INFO));

    /* Add a non-named int4 field, and a named string field */
    TipcMsgAppendInt4(msg, 5);
    TipcMsgAddNamedStr(msg, "string one", "hello");

    /* Now get the string field */
    TipcMsgGetNamedStr(msg, "string one", &str);
    TutOut("named string field is %s\n", str);

    /* Rewind the index back to the first field, and get the int4 field */
    TipcMsgSetCurrent(msg, 0);
    TipcMsgNextInt4(msg, &i);
    TutOut("first field is %d\n", i);
/*
    * Get the string field again. Note that we don't have to use the
    * name to get it, it's still just an indexed field, like any other.
    */
    TipcMsgNextStr(msg, &str);
    TutOut("second field is %s\n", str);
/*
    * Rewind the index pointer again, and we can "name" the int4 field.
    */
    TipcMsgSetCurrent(msg, 0);
    TipcMsgSetNameCurrent(msg, "int4 zero");
/*
    * We can also get the name of the current field.
    */
    TipcMsgGetNameCurrent(msg, &str);
    TutOut("name of first field is %s\n", str);
} /* main */
```

Destroying a Message

When a process finishes with a message, the message can be destroyed to free up the memory it occupies. This is accomplished using the `TipcMsgDestroy` function. For example:

```
if (!TipcMsgDestroy(msg)) {
    /* error */
}
```

`TipcMsgDestroy` decrements the message reference count and destroys the message if the reference count is zero. `TipcMsgDestroy` can also call `TipcMsgAck` to acknowledge the message for GMD. For a detailed discussion of GMD, see Chapter 4, Guaranteed Message Delivery.

Sometimes it is difficult to know when `TipcMsgDestroy` should be used and when it shouldn't. Many of the API functions discussed in Chapter 2, Connections, give access to messages at various points of operation, and `TipcMsgDestroy` must be used with care with these functions. The following simple model can be used to clarify message memory management: most functions do not change who owns the message and who is responsible for destroying it, but some functions give away destroy responsibility and a few functions take away destroy responsibility. In other words, some functions require the caller to call `TipcMsgDestroy` later, and some functions require the caller not to call `TipcMsgDestroy`.

These functions give destroy responsibility to the caller:

- `TipcMsgCreate`
- `TipcMsgIncrRefCount`
- `TipcMsgClone`
- `TipcMsgFileRead`
- `TipcConnMsgNext`
- `TipcConnMsgSearch`
- `TipcConnMsgSearchType`
- `TipcConnMsgSendRpc`
- `TipcSrvMsgNext`
- `TipcSrvMsgSearch`
- `TipcSrvMsgSearchType`
- `TipcSrvMsgSendRpc`

These functions take destroy responsibility from the caller:

- `TipcMsgDestroy`
- `TipcConnMsgInsert`
- `TipcSrvMsgInsert`

Reusing a Message

If a process is finished with a message, but will be constructing other messages in the near future, the message can simply be reused. All of the non-data properties can be set to other values if necessary, and the function `TipcMsgSetNumFields` can be used to remove all existing fields from the messages. For example:

```
if (!TipcMsgSetNumFields(msg, 0)) {  
    /* error */  
}
```

Advanced Uses of Messages

What we covered so far are the most common ways of working with messages. There are many other ways, and some of these advanced uses include:

- Checking the Type of the Current Field
- Cloning a Message
- Array Fields
- Array Fields
- Constructing a Message Within a Message
- Pointer Fields
- Unknown Field Values
- High Performance Guidelines

Checking the Type of the Current Field

The function `TipcMsgNextType` can be used to look at the type of the current field without advancing the current field. This is useful when the type of the current field can vary.

For example, if the current field is either a STR or REAL8 field, this code is used to access the field:

```
T_IPC_FT ft;
T_STR str;
T_REAL8 real8;

if (!TipcMsgNextType(msg, &ft)) {
    /* error */
}
switch (ft) {
case T_IPC_FT_STR:
    if (!TipcMsgNextStr(msg, &str)) {
        /* error */
    }
    break;
case T_IPC_FT_REAL8:
    if (!TipcMsgNextReal8(msg, &real8)) {
        /* error */
    }
    break;
default:
    /* error */
}
```

Cloning a Message

Before destroying a message, you may want to make a copy of the message to use for other purposes. The clone is an identical copy of the original message, but does not share the memory of the original message. A message is cloned using the `TipcMsgClone` function. For example:

```
clone_msg = TipcMsgClone(msg);
if (clone_msg == NULL) {
    /* error */
}
```

Array Fields

Arrays can also be fields in a message. Array fields are appended to a message with the functions `TipcMsgAppendTypeArray`, where *Type* is the type of the array. These append functions also require the number of elements in the array as a parameter. For example:

```
T_REAL8 real8_array[10]; /* can hold up to 10 values */

real8_array[0] = 3.0;
real8_array[1] = 4.0;
real8_array[2] = 55.75;
if (!TipcMsgAppendReal8Array(msg, real8_array, 3)) {
    /* error */
}
```



Empty arrays, where the number of array elements is zero, are allowed.

When using array fields with `TipcMsgWrite`, both the array and the number of elements must be supplied. For example, the above array field could be appended as follows:

```
if (!TipcMsgWrite(msg,
                  T_IPC_FT_REAL8_ARRAY, real8_array, 3, NULL)) {
    /* error */
}
```

Array fields are accessed from a message with the functions `TipcMsgNextTypeArray`, where *Type* is the type of the array. These append functions also require a parameter where they store the number of elements in the array. For example:

```
T_REAL8 *real8_array;
T_INT4 array_size;

if (!TipcMsgNextReal8Array(msg, &real8_array, &array_size)) {
    /* error */
}
```

When using array fields with `TipcMsgRead`, storage for both the array and the number of elements must be supplied. For example, the above array field could be accessed as follows:

```
if (!TipcMsgRead(msg,
                  T_IPC_FT_REAL8_ARRAY, &real8_array, &array_size,
                  NULL)) {
    /* error */
}
```

Constructing a Message Within a Message

Messages can also be fields in a message. This is useful for batching several messages into one large transaction or when using a message as a container for another message, such as the `GMD_FAILURE` message type. Messages are appended as fields with the function `TipcMsgAppendMsg`. When a message is appended as a field to another message, a complete copy of the field message is made. For example:

```
if (!TipcMsgAppendMsg(msg, field_msg)) {
    /* error */
}
if (!TipcMsgDestroy(field_msg)) {
    /* error */
}
```

Once `TipcMsgAppendMsg` returns, changing (or even destroying) the message used as a field does not affect the other message. Message fields are accessed from a message with the function `TipcMsgNextMsg`. For example:

```
T_IPC_MSG field_msg;
if (!TipcMsgNextMsg(msg, &field_msg)) {
    /* error */
}
```

Pointer Fields

Normally, when a field is appended to a message the field data is copied into the message. For large fields, such as a 10-megabyte binary image, this memory copy can decrease performance measurably. To avoid this, all array-oriented fields, including string, message, and binary fields, can be appended to a message as a pointer field. A pointer field does not make a copy of the data. Instead, the supplied pointer is entered directly into the new message field's internal data structure. When using pointer fields, the field data must not be deallocated or changed while the message field is still valid.

Pointer fields are appended to a message with the functions `TipcMsgAppendTypePtr`, where *Type* is the field type. These pointer field construction functions are available:

- `TipcMsgAppendBinaryPtr`
- `TipcMsgAppendInt2ArrayPtr`
- `TipcMsgAppendInt4ArrayPtr`
- `TipcMsgAppendInt8ArrayPtr`
- `TipcMsgAppendMsgPtr`
- `TipcMsgAppendMsgArrayPtr`
- `TipcMsgAppendReal4ArrayPtr`
- `TipcMsgAppendReal8ArrayPtr`
- `TipcMsgAppendReal16ArrayPtr`
- `TipcMsgAppendStrPtr`
- `TipcMsgAppendStrArrayPtr`
- `TipcMsgAppendXmlPtr`

The `TipcMsgAppendTypePtr` functions have one extra parameter in addition to the parameters for the `TipcMsgAppendType` functions. This extra parameter, an optional `T_IPC_MSG_FIELD` message field, is filled in if non-null. For example:

```
T_REAL8 real8_array[10]; /* can hold up to 10 values */
T_IPC_MSG_FIELD msg_field;

real8_array[0] = 3.0;
real8_array[1] = 4.0;
real8_array[2] = 55.75;
if (!TipcMsgAppendReal8ArrayPtr(msg, real8_array, 3, &msg_field))
{
    /* error */
}
```

Another advantage of pointer fields is that they can be easily resized with the function `TipcMsgFieldSetSize` after the field is created. This allows message fields to be efficiently and easily resized. For example:

```
/* add an array element and resize the pointer field size */
real8_array[3] = 318.9;
if (!TipcMsgFieldSetSize(msg_field, 4)) {
    /* error */
}
```

Once a pointer field is appended to a message and the message is sent through a connection, the pointer field data is copied into the connection just like a non-pointer field. Pointer fields can also be added by name. Pointer fields are added by name to a message with the functions `TipcMsgAddNamedTypePtr`, where *Type* is the field type. These pointer field construction functions are available:

- `TipcMsgAddNamedBinaryPtr`
- `TipcMsgAddNamedGuidArrayPtr`
- `TipcMsgAddNamedInt2ArrayPtr`
- `TipcMsgAddNamedInt4ArrayPtr`
- `TipcMsgAddNamedInt8ArrayPtr`
- `TipcMsgAddNamedMsgPtr`
- `TipcMsgAppendMsgArrayPtr`
- `TipcMsgAddNamedReal4ArrayPtr`
- `TipcMsgAddNamedReal8ArrayPtr`
- `TipcMsgAddNamedReal16ArrayPtr`
- `TipcMsgAddNamedStrPtr`
- `TipcMsgAddNamedStrArrayPtr`
- `TipcMsgAddNamedXmlPtr`

Unknown Field Values

In some cases, it is necessary to designate a field value “unknown” or “invalid”. This may be used in cases where data is missing, say in a sequence of time series data. For some data types, like `T_STR`, this can be accomplished using an invalid value as a marker (such as `NULL`). However, some data types, such as numeric values, do not allow any invalid values, even for use as a marker. For these cases, there are a few special message field functions:

- `TipcMsgAppendUnknown`
- `TipcMsgNextUnknown`
- `TipcMsgGetCurrentFieldKnown`

These functions allow any field to be marked uniquely as holding an unknown value. Fields with unknown values still have a type, so `TipcMsgAppendUnknown` takes a field type parameter:

```
TipcMsgAppendUnknown(msg, T_IPC_FT_INT4);
```

If the type of the unknown field is significant, it can be accessed with `TipcMsgNextType`, like any other field. Using a normal `TipcMsgNext*` function to access a field with an unknown value will fail. If `TipcMsgNextInt4` were used on the above field, it would return `FALSE`, and no value would be returned. Instead, `TipcMsgNextUnknown` can be used to access unknown fields, or the field pointer could simply be set to skip the field, using `TipcMsgSetCurrent`. To determine if the current field value is unknown, use `TipcMsgGetCurrentFieldKnown`.

High Performance Guidelines

There are two simple recommended strategies to use when sending large amounts of data with messages:

- When constructing messages with many values of the same type, use array fields instead of non-array fields. For example, a single `INT4_ARRAY` field with 10,000 elements uses much less memory than 10,000 `INT4` fields.
- When constructing large fields, use pointer fields instead of non-pointer fields to avoid a memory copy. For example, a 10,000 element `INT4_ARRAY` pointer field uses less CPU time than a normal 10,000 element `INT4_ARRAY` field.
- When sending messages with very large payloads or text, such as XML, use message compression. For more information on compression, see Message Compression on page 271.

Message Files

A message file is a file (in either text format or binary format) containing one or more messages. It provides a means to capture real or simulated data and is typically used as a testing and debugging tool by any process in a SmartSockets application. A message file serves these purposes:

- It simulates real data, allowing project development to proceed without tying into real-time data.
- It allows a process to be tested as a stand-alone component before beginning integrated testing of the project.
- It captures real data that can be used to debug a process.
- It provides a method for testing and training operators in response to events.

A message file typically has a `.msg` file extension (although SmartSockets does not enforce this extension).

Text Message Files

Messages are printed in a text message file in the following format:

message_type destination data

The sender, priority, delivery mode, reference count, sequence number, and user-defined property of a message are not included. To be written to a message file, the message *data* must be written according to the message type grammar (see Grammar on page 32 for more information on message type grammars). Groups of fields must be delimited by curly braces (`{}`), unless the group is not repeated, in which case the curly braces can be omitted. Array fields, binary fields, and messages whose type has a grammar of `verbose` must also be delimited by curly braces (`{}`). Comments (delimited by `/* */`, `(*)`, or `//` and end-of-line) are allowed anywhere in the file. If you are creating a text message file manually, with a text editor, an easy way to check how to format the file is to use `TpcMsgFileWrite` in a small program to see how it formats a similar message.

This is an example of a text message file holding four messages:

```
numeric_data thermal {
    voltage 33.4534
    switch_pos 0
}
numeric_data eps temperature 200.4 // note: no curly braces needed
/* sent to the operator's display */
info _hci "Satellite has entered science mode."
string_data thermal relay1 "off"
```

Binary Message Files

A binary message file has both advantages and disadvantages when compared to text message files. The advantages include:

- All message properties except reference count and delivery timeout are included.
- Reading and writing are faster than with text message files (no text formatting).
- Binary messages are encoded so that they can be properly read on platforms other than the ones on which they were written.

Binary message files, however, are not easily editable with a text editor.

Using Message Files

Message files are created with the function `TipcMsgFileCreate` and can be created for reading (an existing file is opened), for writing in text format, for writing in binary format, or for appending. `TipcMsgFileCreate` automatically detects the proper format (text or binary) when used for reading and appending. For example:

```
msg_file = TipcMsgFileCreate("output.msg",
T_IPC_MSG_FILE_CREATE_WRITE);
if (msg_file == NULL) {
    /* error */
}
```

Messages are written to a message file using the `TipcMsgFileWrite` function. For example:

```
if (!TipcMsgFileWrite(msg_file, msg)) {
    /* error */
}
```

Any process can read messages from the message file using the `TipcMsgFileRead` function. The message file must be created for reading. For example:

```
msg_file = TipcMsgFileCreate("input.msg",
T_IPC_MSG_FILE_CREATE_READ);
if (msg_file == NULL) {
    /* error */
}
if (!TipcMsgFileRead(msg_file, &msg)) {
    /* error */
}
```



Because `T_IPC_MSG_FILE_CREATE_READ` was used in this example, the actual file must already exist. The structure needed for the message file is created in an existing file. If no file exists, you get a null pointer error message.

Messages can also be logged into a message file from an RTclient process. See Message File Logging Categories on page 211 for more information.

When a process finishes with a message file, the message file can be destroyed to free up the memory it uses. For example:

```
if (!TipcMsgFileDestroy(msg_file)) {
    /* error */
}
```

TipcMsgFileDestroy does not remove the file on disk, only the in-memory record of the file.

Advanced Use of Message Files

It is possible to serialize messages into a binary buffer, and retrieve them from the buffer. Here is a sample program that illustrates this technique:

```
#include <rtworks/ipc.h>

char *filename = "file";

void SerializeMsgToDisk(T_IPC_MSG msg)
{
    T_BUF buf;
    T_PTR data;
    T_INT4 size;
    FILE *fp;

    buf = TutBufCreate(128);
    TipcBufMsgAppend(buf, msg);
    TutBufGetSize(buf, &size);

    data = TutBufNextAligned(buf, size, 1);
    if (data == NULL) {
        /* error */
    }

    fp = fopen(filename, "w");
    if (fp == NULL) {
        /* error */
    }

    fwrite(data, sizeof(T_UCHAR), size, fp);
    fclose(fp);
    TutBufDestroy(buf);
}
```

```

T_IPC_MSG ReadFromDisk(void)
{
    T_BUF buf;
    T_PTR data;
    T_INT4 size;
    T_BOOL status;
    FILE *fp;
    T_IPC_MSG msg;

    status = TutFileGetSize(filename, &size);
    if (status == FALSE) {
        /* error */
    }

    fp = fopen(filename, "r");
    if (fp == NULL) {
        /* error */
    }

    T_MALLOC(data, size, T_PTR);
    fread(data, sizeof(T_UINT1), size, fp);

    /* create static buffer, set max size */
    buf = TutBufCreateStatic(data, size);

    /* set write pointer size */
    status = TutBufSetSize(buf, size);

    msg = TipcBufMsgNext(buf);
    if (msg == NULL) {
        /* error */
    }

    TutBufDestroy(buf);
    T_FREE(data);

    return msg;
}

int main(int argc, char *argv[])
{
    T_IPC_MSG msg_in, msg_out;

    msg_in = TipcMsgCreate(TipcMtLookupByNum(T_MT_INFO));
    TipcMsgAppendStr(msg_in, "hello world");

    SerializeMsgToDisk(msg_in);
    TipcMsgDestroy(msg_in);

    msg_out = ReadFromDisk();
    TipcMsgPrint(msg_out, TutOut);
    TipcMsgDestroy(msg_out);
}

```


Chapter 2 **Connections**

Messages are packets of information that are the basis for all interprocess communication in SmartSockets. All messages are transmitted between processes through connections. A connection is an endpoint of a direct communication link used to send and receive messages between two processes. The two processes, called peer processes, share the link. However, each process has a unique endpoint that it can manipulate independently. Connections can operate on messages in many different ways.

Topics

- *Features of Connections, page 70*
- *Connection Composition, page 71*
- *Sockets, page 85*
- *Working With Connections, page 89*
- *Using Threads With Connections, page 125*
- *Advanced Uses of Connections, page 141*
- *Handling Network Failures, page 149*

Features of Connections

Connections offer these features:

- Connections can use one of several different interprocess communications (IPC) protocols to transfer messages: TCP/IP or local (non-network).
- Connections can execute callback functions at various points while operating on messages to perform user-defined actions.
- Connections can detect many kinds of network failures and take steps to recover from these failures.
- Connections can be used for request-reply communication by sending and receiving messages.
- Connections can send messages with or without guaranteed message delivery (GMD). GMD stores copies of messages in files to enable total recovery from network failures.
- Connections are safe to use in multithreaded programs.

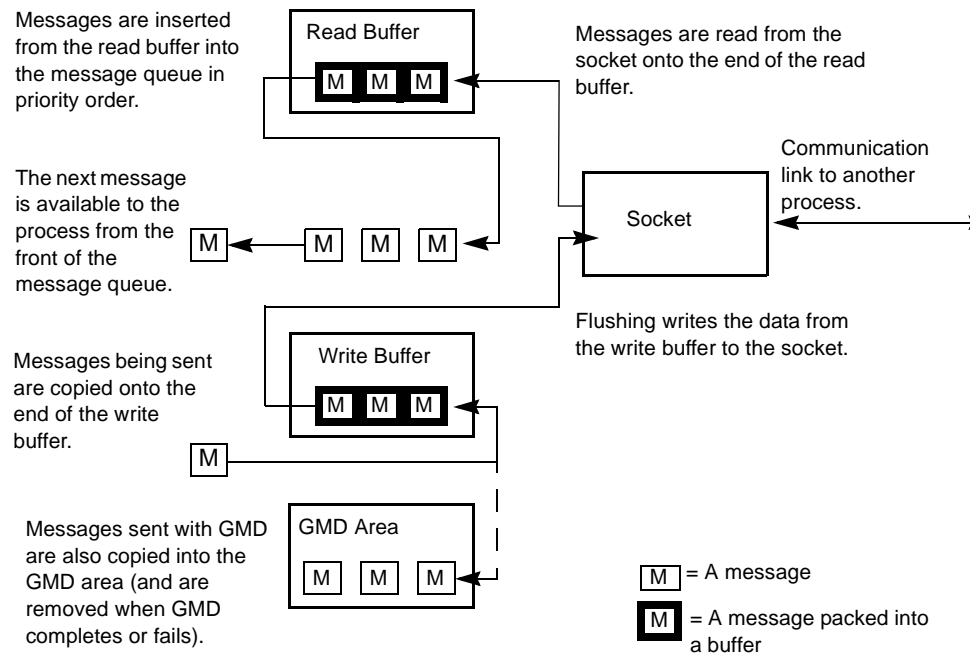
Many advanced features of SmartSockets are message-based services available through connections. Connections and messages are flexible, and SmartSockets uses this flexibility to transmit not only user application messages but also many internal SmartSockets messages. The largest example of a message-based service is monitoring, described in Chapter 5, Project Monitoring. GMD is also implemented using internal messages extensively.

Connection Composition

A connection (C type T_IPC_CONN) is composed of several parts, or properties. Not all properties are directly accessible, but all are important.

Figure 2 shows the flow of messages through the relevant properties of a connection.

Figure 2 The Flow of Messages Through a Connection



A connection has these properties:

Socket	operating system device that provides the communication link
Read Buffer	where incoming messages are first stored when they are read in
Write Buffer	where outgoing messages are stored before they are written
Message Queue	where incoming messages are stored before being processed

Block Mode	whether or not to wait for read and write operations to complete
Auto Flush Size	how many bytes of outgoing messages to allow to accumulate
Read Timeout	how often messages are expected to be available for reading
Write Timeout	how often messages are expected to be able to be written
Keep Alive Timeout	how long to wait for a keep alive query to complete
Delivery Timeout	how long to wait for GMD to complete
GMD Area	holds GMD information for both incoming and outgoing messages
Thread Synchronization	protection for multithreaded access using read, write, process, GMD, and queue mutexes
Peer Information	information about peer process, such as architecture, node, process ID, unique subject, and user

Socket

The socket property identifies the operating system device that provides the communication link to another process. All messages that are sent and received through the connection are transmitted using the socket. The socket property is an operating system file descriptor of type C type T_INT4. The property is set automatically when a client or server connection is created with `TipcConnCreateClient`, `TipcConnCreateServer`, or `TipcConnAccept`. To change the value of the property, you can use `TipcConnSetSocket`:

```
if (!TipcConnSetSocket(conn, fd)) {
    /* error */
}
```

To get the existing value of the socket property, use `TipcConnGetSocket`:

```
if (!TipcConnGetSocket(conn, &fd)) {
    /* error */
}
```

For an in-depth discussion of sockets, see [Sockets](#) on page 85.

A concept related to the connection socket is the connection Xt-compatible source. See [Mixing Connections and Xt Intrinsics \(Motif\)](#) on page 141 for more information on how to use connections in a Motif program.

Read Buffer

The read buffer of a connection is where incoming messages are first stored when they are read from the connection's socket. The incoming messages arrive as a stream of bytes, which are unpacked into messages and inserted into the connection's message queue (for a discussion of connection message queues, see [Message Queue](#) on page 74) in priority order. Each time data is read from the connection's socket, only a piece of a message may arrive. The read buffer is used to reassemble all the pieces of an incoming message.

The read buffer is not directly accessible. The read buffer is sized dynamically to hold all incoming data and is limited only by the amount of available virtual memory. Each time data is read from the connection's socket, all complete messages in the read buffer are unpacked and moved from the read buffer to the message queue. Thus the read buffer does not use a large amount of memory unless large messages are used.

The function `TipcConnRead` reads data from the connection's socket into the read buffer and converts the data into messages, which are then inserted into the connection's message queue with the function `TipcConnMsgInsert`.

Write Buffer

The write buffer of a connection is where outgoing messages are stored before they are written to the connection's socket. By accumulating several messages in the write buffer before actually writing them to the socket, better performance can be achieved.

The write buffer is not directly accessible. The write buffer is sized dynamically to hold all buffered outgoing data and is limited only by the amount of available virtual memory. The connection property `auto flush size` helps to limit the amount of memory used by the write buffer. For a discussion of the connection `auto flush size`, see [Auto Flush Size](#) on page 76.

The function `TipcConnMsgSend` copies a message to the end of the write buffer, and the function `TipcConnFlush` writes the write buffer to the connection's socket.

Message Queue

The message queue of a connection is where incoming messages are stored waiting to be processed. Messages are normally stored in the message queue ordered by message priority, although they can be inserted anywhere into the message queue.

The message queue is not directly accessible. The function `TipcConnMsgInsert` inserts a message into a connection's message queue. The function `TipcConnMsgNext` gets the first message from a connection's message queue. The function `TipcConnMsgSearch` searches a connection's message queue for a specific message. The function `TipcConnGetNumQueued` gets the number of messages in a connection's message queue.

Block Mode

The block mode property is a boolean that identifies whether or not a process waits for a read, write, or accept operation to complete on a connection's socket. Read, write, and accept operations are treated slightly differently. Read operations are always given a certain period of time to complete, while write operations are not. A write or accept operation either completes immediately or the process waits until the operation completes.

To change the value of the block mode property, use `TipcConnSetBlockMode`:

```
if (!TipcConnSetBlockMode(conn, TRUE)) {
    /* error */
}
```

To get the existing value of the property, use `TipcConnGetBlockMode`:

```
if (!TipcConnGetBlockMode(conn, &block_mode)) {
    /* error */
}
```

If the block mode property is `TRUE`, which is the default, the function `TipcConnRead` does not return until it has read some data from the connection's socket into the connection's read buffer, an error has occurred, or the specified period of time has elapsed. The function `TipcConnFlush` does not return until it has written all data from the connection's write buffer to the connection's socket or an error has occurred. The function `TipcConnAccept` does not return until it has accepted a client connection or an error has occurred.

If the block mode property is `FALSE`, then non-blocking read, write, and accept operations are enabled. It is unusual to use non-blocking accept operations; they are provided for completeness only. The behavior of non-blocking read and write operations depends on the settings of the connection timeout properties (see `Read Timeout` on page 77, `Write Timeout` on page 77, and `Keep Alive Timeout` on page 78 for more information on these timeout properties). A connection's block

mode must be `FALSE` for the timeout properties to have any effect. Table 3 shows the relationship between the block mode and timeout properties. If the block mode property is `FALSE` and the write timeout property is `0.0`, then `TipcConnFlush` returns immediately if not all data can be written. If the block mode property is `FALSE` and the write timeout property is greater than `0.0`, then `TipcConnFlush` does not return until all data has been written or a period of time equal to the write timeout property has elapsed.

Table 3 Relationship Between Connection Block Mode and Timeout Properties

Block Mode Property	Timeout Properties	Effect on Read and Write Operations
TRUE	0.0	Read operations can block for a certain period of time, specified by the <i>timeout</i> parameter to <code>TipcConnRead</code> . Write operations can block indefinitely.
TRUE	greater than 0.0	Read operations can block for a certain period of time, specified by the <i>timeout</i> parameter to <code>TipcConnRead</code> . Write operations can block indefinitely. Timeouts greater than 0.0 have no effect when block mode is <code>TRUE</code> .
FALSE	0.0	Read operations can block for a certain period of time, specified by the <i>timeout</i> parameter to <code>TipcConnRead</code> . Write operations can never block.
FALSE	greater than 0.0	Read operations can block for a certain period of time, specified by the <i>timeout</i> parameter to <code>TipcConnRead</code> , but are limited to the value of the connection read timeout property. Write operations can block for a certain period of time, but are limited to value of the write timeout connection property.

Auto Flush Size

The auto flush size property is defined as a four-byte integer that identifies how many bytes of data are allowed to accumulate in a connection's write buffer before it is automatically written (flushed) to the connection's socket. When a message is copied to the end of the connection's write buffer with the function `TipcConnMsgSend`, the auto flush size is checked, and `TipcConnMsgSend` calls the function `TipcConnFlush` if the number of bytes of data in the write buffer is greater than the auto flush size. In addition, both `TipcConnRead` and `TipcConnCheck` automatically flush the write buffer to the socket when reading and checking for reading.

If the auto flush size property of a connection is set to 0, then each outgoing message is immediately written to the connection's socket. To enable infinite buffering, an auto flush size of `T_IPC_NO_AUTO_FLUSH` can be used, and outgoing messages are never automatically flushed (they have to be explicitly flushed). The default size is 8192 bytes.

To change the value of the auto flush size property, use `TipcConnSetAutoFlushSize`:

```
if (!TipcConnSetAutoFlushSize(conn, 0)) {
    /* error */
}
```

To get the existing value for the property, use `TipcConnGetAutoFlushSize`:

```
if (!TipcConnGetAutoFlushSize(conn, &auto_flush_size)) {
    /* error */
}
```

Read Timeout

The read timeout property is defined as an eight-byte real number that identifies how often, in seconds, data is expected to be available for reading on a connection's socket. This timeout is used to check for possible network failures.

Whenever a process is waiting for a read operation on a connection's socket to complete, it does not wait longer than *read timeout* seconds past the time of the last successful read. If a period of time longer than the read timeout has elapsed, then a hardware or software failure may have occurred; a query then is sent to the other end of the connection to determine if the connection is still alive. This query is called a keep alive. See Handling Network Failures on page 149 for more information on keep alives.

If the read timeout property of a connection is set to 0.0, which is the default, then checking for read timeouts is disabled. Setting the block mode property of a connection to TRUE also disables checking for read timeouts.

To change the value of the read timeout property, use `TipcConnSetTimeout`:

```
if (!TipcConnSetTimeout(conn, T_IPC_TIMEOUT_READ, 10.0)) {
    /* error */
}
```

To get the existing value for the property, use `TipcConnGetTimeout`:

```
if (!TipcConnGetTimeout(conn, T_IPC_TIMEOUT_READ, &read_timeout))
{
    /* error */
}
```

Write Timeout

The write timeout property is defined as an eight-byte real number that identifies how often, in seconds, data is expected to be able to be written to a connection's socket. This timeout is used to check for possible network failures.

Whenever a process is waiting for a write operation on a connection's socket to complete, it does not wait longer than *write timeout* seconds past the time of the last successful write. If a period of time longer than the write timeout has elapsed, then a hardware or software failure may have occurred, and the error callbacks for the connection will be called to recover from the error. See Error Callbacks on page 110 for more information on error callbacks. Unlike the read timeouts, where a keep alive is initiated, no keep alive is initiated for write timeouts. See Handling Network Failures on page 149 for more information on keep alives.

If the write timeout property of a connection is set to 0.0, which is the default, checking for write timeouts is disabled. Setting the block mode property of a connection to TRUE also disables checking for write timeouts.

To change the value for the write timeout property, use `TipcConnSetTimeout`:

```
if (!TipcConnSetTimeout(conn, T_IPC_TIMEOUT_WRITE, 10.0)) {
    /* error */
}
```

To get the existing value for the property, use `TipcConnGetTimeout`:

```
if (!TipcConnGetTimeout(conn, T_IPC_TIMEOUT_WRITE,
    &write_timeout)) {
    /* error */
}
```

Keep Alive Timeout

The keep alive timeout property is defined as an eight-byte real number that identifies how long, in seconds, to wait for a keep alive query to complete. A keep alive query consists of sending a `KEEP_ALIVE_CALL` through a connection and waiting for the process at the other end to send back a `KEEP_ALIVE_RESULT` message to indicate that the connection is still alive. See [Handling Network Failures](#) on page 149 for more information on the keep alive timeout and keep alives.

If the keep alive timeout property of a connection is set to 0.0, which is the default, then keep alive queries are disabled. To change the value, use `TipcConnSetTimeout`:

```
if (!TipcConnSetTimeout(conn, T_IPC_TIMEOUT_KEEP_ALIVE, 10.0)) {
    /* error */
}
```

To get the existing value for the property, use `TipcConnGetTimeout`:

```
if (!TipcConnGetTimeout(conn, T_IPC_TIMEOUT_KEEP_ALIVE,
    &keep_alive_timeout)) {
    /* error */
}
```


Delivery Timeout

The delivery timeout property is defined as an eight-byte real number that identifies how long, in seconds, to wait for guaranteed delivery of a message sent from this process through a connection. This timeout is used to check for possible network failures, although at a slightly different level from the read timeout, write timeout, and keep alive timeout (those three are not directly involved with GMD).

Whenever a process sends a message with a delivery mode of `T_IPC_DELIVERY_SOME` or `T_IPC_DELIVERY_ALL`, a copy of the message and the current wall clock time are saved in the connection GMD area. See GMD Area on page 80 for more information on GMD areas. The message copy is removed when acknowledgment of delivery is received by the sender from the receiving process(es). Delivery timeouts are checked each time the sending process reads data or checks if data can be read from the connection. If *delivery timeout* seconds have elapsed since the message was sent with GMD, then a GMD failure has occurred, and a `GMD_FAILURE` message will be processed by the sender. See Chapter 4, Guaranteed Message Delivery for more information on GMD. Note that each message can also have its own delivery timeout, which defaults to the connection's delivery timeout if it is not set before the message is sent.

If the delivery timeout property of a message is set to `0.0`, which is the default, checking for delivery timeouts is disabled. To change the value, use `TipcConnSetTimeout`:

```
if (!TipcConnSetTimeout(conn, T_IPC_TIMEOUT_DELIVERY, 10.0)) {
    /* error */
}
```

To get the existing value of the property, use `TipcConnGetTimeout`:

```
if (!TipcConnGetTimeout(conn, T_IPC_TIMEOUT_DELIVERY,
                        &delivery_timeout)) {
    /* error */
}
```

GMD Area

The GMD area for a connection holds guaranteed message delivery information for both incoming and outgoing messages. There are two types of GMD:

- file-based GMD

File-based GMD stores GMD information in files to enable recovery if the program crashes and is restarted. For file-based GMD, the GMD area is in a directory on disk. File-based GMD is the default and provides the most reliable GMD in the event of a system failure. However, because it writes the GMD messages to disk, it is slower than memory-based GMD.

- memory-based GMD

Memory-based GMD stores GMD information in a GMD area that is held in memory and is faster than file-based GMD. It protects your messages against network failures and lost connections. However, if a system failure wipes out memory, such as when a program crashes and restarts, the GMD messages stored in memory in the GMD area are lost.

The GMD area is not directly accessible. Using the `RTclient` option `Ipc_Gmd_Directory`, you can specify where you want the file-based GMD area created. Once the GMD area is created, it cannot be changed or destroyed except by destroying the connection. The function `TipcConnSetGmdMaxSize` can be used to set the maximum size of the GMD area.

See Chapter 4, *Guaranteed Message Delivery* for more information on GMD.

Thread Synchronization

The set of connection properties used for thread synchronization protect connections for multithreaded access. There are no functions that get or set the values for these properties:

- **Read Mutex**

The read mutex property protects operations that access the connection's message queue, read buffer, or GMD high sequence number table, or read data from the connection's socket.

- **Write Mutex**

The write mutex property protects operations that access the connection's write buffer or write data to the connection's socket.

- **Process Mutex**

The process mutex property protects operations that access the connection's process and default callback lists. A read/write mutex is used here so that multiple threads may execute a connection's process and default callbacks concurrently.

- **GMD Mutex**

The gmd mutex property is a temporary mutex that protects operations that access the connection's GMD area. A temporary mutex means that this mutex is never held for an indefinite period of time.

- **Queue Mutex**

The queue mutex property is a temporary mutex that protects operations that access the connection's queue such as a message insert or a message delete. A temporary mutex means this mutex is never held for an indefinite period.

For more information on using connections in multithreaded programs, see [Using Threads With Connections](#) on page 125. For more information on these mutex properties, see [Working With Threads and Connections](#) on page 139.

Peer Information

These connection properties are all pieces of information about the process at the other end of the connection. These properties are all simple types of monitoring and can be useful for diagnostic purposes. For more information on monitoring, see Chapter 5, Project Monitoring.

Architecture

The architecture property is defined as an identifier string that identifies the SmartSockets architecture of the peer process. The architecture is in the form *Machine_OperatingSystem* (for example, *sun4_solaris*).

This value is stored in:

UNIX the environment variable RTARCH

OpenVMS the logical RTARCH

Windows the environment variable RTARCH

This property is set automatically by `TipcConnCreateClient` and `TipcConnAccept`, and cannot be set manually. To find out the value set for this property, use `TipcConnGetArch`:

```
if (!TipcConnGetArch(conn, &arch)) {
    /* error */
}
```

Node

The node property is defined as a string that identifies the node name of the peer process. This property is set automatically by `TipcConnCreateClient` and `TipcConnAccept`, and cannot be set manually. To find out the value set for this property, use `TipcConnGetNode`:

```
if (!TipcConnGetNode(conn, &node)) {
    /* error */
}
```

Process ID

The process ID property is defined as a four-byte integer that provides the process identifier of the peer process. This property is set automatically by `TipcConnCreateClient` and `TipcConnAccept`, and cannot be set manually. To find out the value set for this property, use `TipcConnGetPid`:

```
if (!TipcConnGetPid(conn, &pid)) {
    /* error */
}
```

Unique Subject

The unique subject property is defined as a string that identifies the unique subject of the peer process. The unique subject, which is used to uniquely identify all SmartSockets processes, is stored in the option `Unique_Subject`. See [Configuring GMD on page 331](#) for a discussion of how `Unique_Subject` is used by connections.

This property is set automatically by `TipcConnCreateClient` and `TipcConnAccept`, and cannot be changed. To find out the value for this property, use `TipcConnGetUniqueSubject`:

```
if (!TipcConnGetUniqueSubject(conn, &unique_subject)) {
    /* error */
}
```

User

The user property is defined as a string that identifies the user name of the peer process. This property is set automatically by `TipcConnCreateClient` and `TipcConnAccept`, and cannot be set manually. To find out the value set for this property, use `TipcConnGetUser`:

```
if (!TipcConnGetUser(conn, &user)) {
    /* error */
}
```

Callbacks

Callbacks are functions that are executed when certain operations occur. Callbacks are conceptually equivalent to dynamically adding a line of code to a program.

These are the types of callbacks available to you:

- process callbacks — executed while processing a message
- default callbacks — executed if no process callbacks exist
- read callbacks — executed when an incoming message is read in
- write callbacks — executed when an outgoing message is sent
- queue callbacks — executed when a message is inserted into or deleted from the message queue
- error callbacks — executed when an unrecoverable error occurs

When a message is sent out by calling `TipcSrvMsgSend`, the callbacks are called in this order:

1. Write callbacks
2. Encode callbacks

For more discussion about callback functions, see [Callbacks](#) on page 107.

Sockets

As described earlier, all messages sent through connections are transmitted using sockets. When you use connections you do not have to fully understand the nuances of sockets, but a general understanding of sockets helps to illustrate the advantages of using connections instead of just sockets. This section gives a brief tutorial on sockets.

Protocols: TCP/IP and Local

A computer network can be viewed as having several layers:

1. Highest Layer: User Applications (RTserver, connection programs, and so on)
2. Middle Layers: Software Protocols (TCP/IP, and so on)
3. Lowest Layer: Hardware Protocols (Ethernet, Token Ring, and so on)

Most computers support Ethernet, some Token ring, and almost all support the widely available Transmission Control Protocol/Internet Protocol (TCP/IP). TCP/IP was developed at the University of California at Berkeley for BSD UNIX. There are several commercial TCP/IP packages available for OpenVMS, such as MultiNet. To support heterogeneous networks using Ethernet or Token Ring, such as Solaris, Windows, or OpenVMS, SmartSockets supports TCP/IP. In fact, within a single application, a process can communicate with different processes using different protocols.

In addition to the above IPC protocols that send data over a network, it is often useful to perform IPC between two processes on the same computer. This can be thought of as a local IPC protocol. SmartSockets IPC also supports this local protocol.

What is a Socket?

A socket is a software interface to an IPC protocol. Each socket uses one specific IPC protocol. A good analogy for the socket is the telephone. A socket allows one process to speak to another process, much like a telephone allows one person to talk to another. There are rules for how the two processes must make the connection (similar to how one must pick up a telephone, dial a number, and have someone answer at the other end).

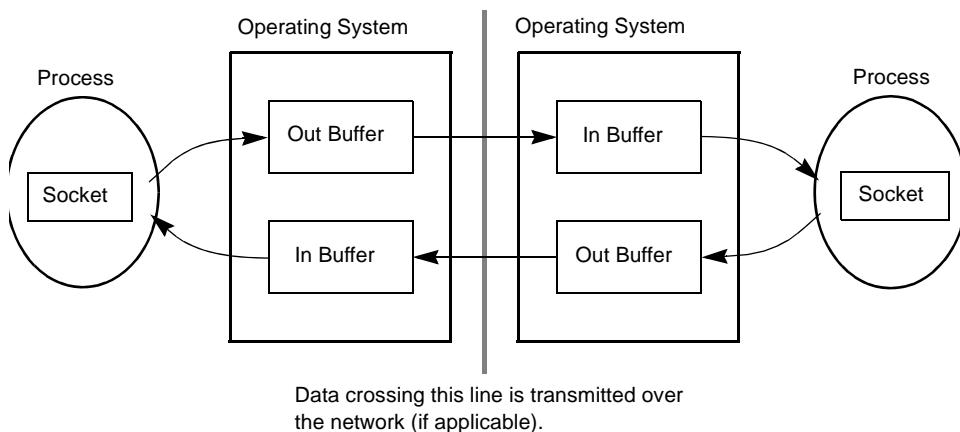
There are several different types of sockets. The most common type is called a stream socket, and it is the type used by SmartSockets IPC. Other kinds of sockets include datagram sockets and raw sockets. From this point on, the term socket is used to mean a stream socket. Unlike a telephone, sockets are truly bidirectional: both processes can write (talk) and read (listen) at the same time.

Most current operating systems support sockets. Sockets are very portable, though there are differences between the various implementations of sockets. None of the OpenVMS implementations of sockets, however, support the TCP/IP and local IPC protocols that SmartSockets can use. On OpenVMS, SmartSockets supplies its own socket functions (which are implemented using native OpenVMS system calls like `SYS$QIO`) to allow TCP/IP and local sockets to be used by the same program. See the *TIBCO SmartSockets Utilities* reference for more information on how to use these socket functions.

How Sockets Work

Each socket has four buffers (one for each process for each direction) associated with it. These buffers are usually at least 8192 bytes each, but the size of each buffer can be changed dynamically. Figure 3 shows these buffers.

Figure 3 Socket Data Buffering



When one process writes data to a socket, the data is copied into the appropriate outgoing buffer, and then the operating system takes care of transmitting the data to the incoming buffer for the receiving process. If the incoming buffer fills up or if there is a hardware or software failure, then the outgoing buffer starts to fill up. If the outgoing buffer is full, the process either waits for space in the buffer or gets an error condition. When the process at the other end of the socket reads data from the socket, the data comes out of the socket's incoming buffer. If the incoming buffer is empty, the process either waits for data to appear or gets an error condition. When a process waits, it blocks (does not use CPU time) until the socket is ready again.

Sockets handle several types of simple network failures, such as reordering data that arrives out of order and resending data that doesn't arrive. They do not take care of recovery from more fatal types of errors, however, such as processes or nodes crashing. See Chapter 4, *Guaranteed Message Delivery* for details on how connections overcome these more serious failures.

Sockets are first in, first out (FIFO) byte streams, just like UNIX files. A byte stream does not distinguish between message boundaries. If one process writes 64 bytes to a socket, a process at the other end could decide to read 32 bytes twice, or 8 bytes 8 times, or even 1 byte 64 times. It is more efficient, however, for the reading process to read 64 bytes one time. Sockets are fairly fast on Ethernet, where data transfer rates of 500 kilobytes/second and up are possible.

Because sockets are byte streams, it is up to the two communicating processes to agree on what the bytes mean: some sort of protocol is necessary. If the two processes are on computers that don't use the same formats for integers, floating-point numbers, or character strings, then it becomes more difficult to pass binary data between the processes.

Sockets have several advantages over other UNIX IPC mechanisms. STREAMS and TLI also provide an interface to IPC protocols, but are not as portable as sockets. Pipes are similar to sockets, but they are unidirectional (data can only be transferred one way) and can only be used between related processes (usually where one process is a child of the other process). They cannot be used between unrelated processes. Shared memory allows unrelated processes on the same machine to communicate, but it is somewhat clumsier to use and does not provide a byte stream. Neither shared memory nor pipes allow processes on different computers to communicate.

Advantages of Connections Over Sockets

Connections are implemented using sockets and take advantage of all the features that sockets have to offer. Connections also have many features that make them easier to use and more powerful than sockets:

- Connections have additional data buffers that can expand to hold large numbers of messages.
- Connections have prioritized message queues.
- Connections handle differences between different socket implementations. For example, the Windows and Solaris Version 2.5.1 socket APIs are not 100% compatible.
- Connections automatically unpack messages read from the byte stream of a socket.
- Connections have callbacks to execute user-defined functions when certain operations occur.
- Connections automatically convert the data within messages that are sent between different types of computers.
- Connections can detect and recover from network failures.
- Connections can perform remote procedure calls by sending and receiving messages.
- Connections have guaranteed message delivery, which enables total recovery from network failures.
- Connections can be used safely in multithreaded programs.

Working With Connections

This section discusses how to create, access, and destroy a connection. To learn more about working with messages, see Chapter 1, Messages. The following example programs show the code used to send messages between two processes through a connection. The programs also show how to use all connection callback types. There are two parts to the example: a server process and a client process. These example programs do not use guaranteed message delivery (refer to Working With GMD on page 322 for complete GMD examples).

The source code files for this example are located in these directories:

UNIX:

`$RTHOME/examples/smrtsock/manual`

OpenVMS:

`RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]`

Windows:

`%RTHOME%\examples\smrtsock\manual`

The online source files have additional `#ifdefs` to provide C++ support. These `#ifdefs` are not shown to simplify the example.

Example 4 Server Source Code

```
* connserv.c -- connections example server */

/*
This server process waits for a client to connect to it, creates some
callbacks, and then loops receiving and processing messages.
*/

#include <rtworks/ipc.h>

/* ===== */
/*..cb_process_numeric_data -- process callback for NUMERIC_DATA */
static void T_ENTRY cb_process_numeric_data(
    T_IPC_CONN conn,
    T_IPC_CONN_PROCESS_CB_DATA data,
    T_CB_ARG arg )
{
    T_STR name;
    T_REAL8 value;

    TutOut("Entering cb_process_numeric_data.\n");
}
```

```

    /* set current field to first field in message */
    if (!TipcMsgSetCurrent(data->msg, 0)) {
        TutOut("Could not set current field of message: error <%s>.\n",
            TutErrStrGet());
        return;
    }

    /* access and print fields */
    while (TipcMsgNextStrReal8(data->msg, &name, &value)) {
        TutOut("%s = %s\n", name, TutRealToStr(value));
    }

    /* make sure we reached the end of the message */
    if (TutErrNumGet() != T_ERR_MSG_EOM) {
        TutOut("Did not reach end of message: error <%s>.\n",
            TutErrStrGet());
    }
} /* cb_process_numeric_data */

/* ===== */
/*..cb_default -- default callback */
static void T_ENTRY cb_default(
    T_IPC_CONN conn,
    T_IPC_CONN_DEFAULT_CB_DATA data,
    T_CB_ARG arg )
{
    T_IPC_MT mt;
    T_STR name;

    TutOut("Entering cb_default.\n");

    /* print out the name of the type of the message */
    if (!TipcMsgGetType(data->msg, &mt)) {
        TutOut("Could not get message type from message: error
        <%s>.\n",
            TutErrStrGet());
        return;
    }
    if (!TipcMtGetName(mt, &name)) {
        TutOut("Could not get name from message type: error <%s>.\n",
            TutErrStrGet());
        return;
    }
    TutOut("Message type name is %s.\n", name);
} /* cb_default */

/* ===== */
/*..cb_read -- read callback */
static void T_ENTRY cb_read(
    T_IPC_CONN conn,
    T_IPC_CONN_READ_CB_DATA data,
    T_CB_ARG arg ) /* really (T_IPC_MSG_FILE) */
{
    TutOut("Entering cb_read.\n");

```

```

    /* print out the message to the message file */
    if (!TipcMsgFileWrite((T_IPC_MSG_FILE)arg, data->msg)) {
        TutOut("Could not write to message file: error <%s>.\n",
            TutErrStrGet());
    }
} /* cb_read */

/* ===== */
/*..cb_queue -- queue callback */
static void T_ENTRY cb_queue(
    T_IPC_CONN conn,
    T_IPC_CONN_QUEUE_CB_DATA data,
    T_CB_ARG arg )
{
    T_IPC_MT mt;
    T_STR name;

    TutOut("Entering cb_queue.\n");

    /* get the name of the type of the message */
    if (!TipcMsgGetType(data->msg, &mt)) {
        TutOut("Could not get message type from message: error
<%s>.\n",
            TutErrStrGet());
        return;
    }
    if (!TipcMtGetName(mt, &name)) {
        TutOut("Could not get name from message type: error <%s>.\n",
            TutErrStrGet());
        return;
    }

    /* print out the position of the message being inserted/deleted */
    TutOut("A message of type %s is being %s at position %d.\n",
        name, data->insert_flag ? "inserted" : "deleted",
        data->position);
} /* cb_queue */

/* ===== */
/*..cb_error -- error callback */
static void T_ENTRY cb_error(
    T_IPC_CONN conn,
    T_IPC_CONN_ERROR_CB_DATA data,
    T_CB_ARG arg )
{
    TutOut("Entering cb_error.\n");
    TutOut("The error number is %d.\n", data->err_num);
} /* cb_error */

```

```

/* ===== */
/*..main -- main program */
int main()
{
    T_IPC_CONN server_conn; /* used to accept client */
    T_IPC_CONN client_conn; /* connection to client */
    T_IPC_MT mt; /* message type for creating callbacks */
    T_IPC_MSG_FILE msg_file; /* message file for printing messages */
    T_IPC_MSG msg; /* message received and processed */

    TutOut("Creating server connection to accept clients on.\n");
    server_conn = TipcConnCreateServer("tcp:_node:5252");
    if (server_conn == NULL) {
        TutOut("Could not create server connection: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    /* accept one client */
    TutOut("Waiting for client to connect.\n");
    client_conn = TipcConnAccept(server_conn);
    if (client_conn == NULL) {
        TutOut("Could not accept client: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    /* destroy server conn: it's not needed anymore */
    TutOut("Destroying server connection.\n");
    if (!TipcConnDestroy(server_conn)) {
        TutOut("Could not destroy server connection: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    /* create callbacks to be executed when certain operations occur */
    TutOut("Create callbacks.\n");

    /* process callback */
    mt = TipcMtLookupByNum(T_MT_NUMERIC_DATA);
    if (mt == NULL) {
        TutOut("Could not look up NUMERIC_DATA msg type: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
    if (TipcConnProcessCbCreate(client_conn, mt,
                                cb_process_numeric_data,
                                NULL) == NULL) {
        TutOut("Could not create NUMERIC_DATA process cb: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
}

```

```

/* default callback */
if (TipcConnDefaultCbCreate(client_conn,
                             cb_default, NULL) == NULL) {
    TutOut("Could not create default cb: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* create a message file to use in read callback */
msg_file = TipcMsgFileCreateFromFile(stdout,
                                       T_IPC_MSG_FILE_CREATE_WRITE);
if (msg_file == NULL) {
    TutOut("Could not create message file from stdout: error
<%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
/* read callback */
if (TipcConnReadCbCreate(client_conn,
                          NULL, /* global callback */
                          cb_read,
                          msg_file) == NULL) {
    TutOut("Could not create read cb: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* queue callback */
if (TipcConnQueueCbCreate(client_conn,
                           NULL, /* global callback */
                           cb_queue,
                           NULL) == NULL) {
    TutOut("Could not create queue cb: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

```

```

/* error callback */
if (TipcConnErrorCbCreate(client_conn,
                          cb_error,
                          NULL) == NULL) {
    TutOut("Could not create error cb: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

TutOut("Read and process all messages.\n");
while ((msg = TipcConnMsgNext(client_conn, T_TIMEOUT_FOREVER))
      != NULL) {
    if (!TipcConnMsgProcess(client_conn, msg)) {
        TutOut("Could not process message: error <%s>.\n",
              TutErrStrGet());
    }
    if (!TipcMsgDestroy(msg)) {
        TutOut("Could not destroy message: error <%s>.\n",
              TutErrStrGet());
    }
}

/* make sure we reached the end of the data */
if (TutErrNumGet() != T_ERR_EOF) {
    TutOut("Did not reach end of data: error <%s>.\n",
          TutErrStrGet());
}
if (!TipcConnDestroy(client_conn)) {
    TutOut("Could not destroy client connection: error <%s>.\n",
          TutErrStrGet());
}

if (!TipcMsgFileDestroy(msg_file)) {
    TutOut("Could not destroy message file: error <%s>.\n",
          TutErrStrGet());
}

TutOut("Server process exiting successfully.\n");
return T_EXIT_SUCCESS; /* all done */
} /* main */

```

Example 5 Client Source Code

```

/* connclnt.c -- connections example client */

/*
The client process connects to the server process and sends two
messages to the server.
*/

#include <rtworks/ipc.h>

```



```

/* ===== */
/*..cb_write -- write callback */
static void T_ENTRY cb_write(
    T_IPC_CONN conn,
    T_IPC_CONN_WRITE_CB_DATA data,
    T_CB_ARG arg ) /*really (T_IPC_MSG_FILE) */
{
    TutOut("Entering cb_write.\n");

    /* print out the message to the message file */
    if (!TipcMsgFileWrite((T_IPC_MSG_FILE)arg, data->msg)) {
        TutOut("Could not write to message file: error <%s>.\n",
            TutErrStrGet());
    }
} /* cb_write */

/* ===== */
/*..main -- main program */
int main()
{
    T_IPC_CONN conn; /* connection to server */
    T_IPC_MT mt; /* message type for creating callbacks and messages */
    T_IPC_MSG_FILE msg_file; /* message file for printing messages */
    T_IPC_MSG msg; /* message to send */

    TutOut("Creating connection to server process.\n");
    conn = TipcConnCreateClient("tcp:_node:5252");
    if (conn == NULL) {
        TutOut("Could not create connection to server: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    /* create callbacks to be executed when certain operations occur */
    TutOut("Create callbacks.\n");

    /* create a message file to use in write callback */
    msg_file = TipcMsgFileCreateFromFile(stdout,
                                          T_IPC_MSG_FILE_CREATE_WRITE);
    if (msg_file == NULL) {
        TutOut("Could not create message file from stdout: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
}

```

```

/* write callback */
if (TipcConnWriteCbCreate(conn,
                          NULL, /* global callback */
                          cb_write,
                          msg_file) == NULL) {
    TutOut("Could not create write cb: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

TutOut("Constructing and sending a NUMERIC_DATA message.\n");
mt = TipcMtLookupByNum(T_MT_NUMERIC_DATA);
if (mt == NULL) {
    TutOut("Could not look up NUMERIC_DATA msg type: error
<%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

msg = TipcMsgCreate(mt);
if (msg == NULL) {
    TutOut("Could not create NUMERIC_DATA message: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

if (!TipcMsgWrite(msg,
                  T_IPC_FT_STR, "voltage",
                  T_IPC_FT_REAL8, 33.4534,
                  T_IPC_FT_STR, "switch_pos",
                  T_IPC_FT_REAL8, 0.0,
                  NULL)) {
    TutOut("Could not append to NUMERIC_DATA msg: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

if (!TipcConnMsgSend(conn, msg)) {
    TutOut("Could not send NUMERIC_DATA message: error <%s>.\n",
           TutErrStrGet());
}

TutOut("Constructing and sending an INFO message.\n");
mt = TipcMtLookupByNum(T_MT_INFO);
if (mt == NULL) {
    TutOut("Could not look up INFO message type: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

```

```

/* just reuse previous message */
if (!TipcMsgSetType(msg, mt)) {
    TutOut("Could not set message type: error <%s>.\n",
           TutErrStrGet());
}
if (!TipcMsgSetNumFields(msg, 0)) {
    TutOut("Could not set message num fields: error <%s>.\n",
           TutErrStrGet());
}

if (!TipcMsgAppendStr(msg, "Now is the time")) {
    TutOut("Could not append fields to INFO message: error
<%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

if (!TipcConnMsgSend(conn, msg)) {
    TutOut("Could not send INFO message: error <%s>.\n",
           TutErrStrGet());
}

if (!TipcConnFlush(conn)) {
    TutOut("Could not flush buffered outgoing msgs: error <%s>.\n",
           TutErrStrGet());
}

if (!TipcMsgDestroy(msg)) {
    TutOut("Could not destroy message: error <%s>.\n",
           TutErrStrGet());
}

if (!TipcConnDestroy(conn)) {
    TutOut("Could not destroy connection: error <%s>.\n",
           TutErrStrGet());
}

if (!TipcMsgFileDestroy(msg_file)) {
    TutOut("Could not destroy message file: error <%s>.\n",
           TutErrStrGet());
}

TutOut("Client process exiting successfully.\n");
return T_EXIT_SUCCESS; /* all done */
} /* main */

```

Compiling, Linking, and Running

To compile, link, and run the example programs, first you must either copy the programs to your own directory or have write permission in these directories:

UNIX:

```
$RTHOME/examples/smrtsock/manual
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]
```

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

Step 1 Compile and link the programs

UNIX:

```
$ rtlink -o connserv.x connserv.c
$ rtlink -o conncnt.x conncnt.c
```

OpenVMS:

```
$ cc connserv.c
$ rtlink /exec=connserv.exe connserv.obj
$ cc conncnt.c
$ rtlink /exec=conncnt.exe conncnt.obj
```

Windows:

```
$ nmake /f consw32m.mak
$ nmake /f concw32m.mak
```

On UNIX the `rtlink` command by default uses the `cc` command to compile and link. To use a C++ compiler or a C compiler with a name other than `cc`, set the environment variable `CC` to the name of the compiler, and `rtlink` then uses this compiler. Use these commands to compile and link on UNIX with the GNU C++ compiler `g++`:

```
$ env CC=g++ rtlink -o connserv.x connserv.c
$ env CC=g++ rtlink -o conncnt.x conncnt.c
```

To run the programs, start the server process first in one terminal emulator window and then the client process in another terminal emulator window.

Step 2 Start the server program in the first window**UNIX:**

```
$ connserv.x
```

OpenVMS:

```
$ run connserv.exe
```

Windows:

```
$ connserv.exe
```

Step 3 Start the client program in the second window**UNIX:**

```
$ conncInt.x
```

OpenVMS:

```
$ run conncInt.exe
```

Windows:

```
$ conncInt.exe
```

The output from the server program is:

```
Creating server connection to accept clients on.
Waiting for client to connect.
Destroying server connection.
Create callbacks.
Read and process all messages.
Entering cb_read.
numeric_data _null {
  voltage 33.4534
  switch_pos 0
}
Entering cb_queue.
A message of type numeric_data is being inserted at position 0.
Entering cb_read.
info _null "Now is the time"
Entering cb_queue.
A message of type info is being inserted at position 1.
Entering cb_queue.
A message of type numeric_data is being deleted at position 0.
Entering cb_process_numeric_data.
voltage = 33.4534
switch_pos = 0
Entering cb_queue.
```

```

A message of type info is being deleted at position 0.
Entering cb_default.
Message type name is info.
Entering cb_error.
The error number is 10.
Server process exiting successfully.

```

The output from the client program is:

```

Creating connection to server process.
Create callbacks.
Constructing and sending a NUMERIC_DATA message.
Entering cb_write.
numeric_data _null {
    voltage 33.4534
    switch_pos 0
}
Constructing and sending an INFO messages.
Entering cb_write.
info _null "Now is the time"
Client process exiting successfully.

```

Include Files

Code written in C or C++ that uses the SmartSockets Application Programming Interface (API) must include the header file `<rtworks/ipc.h>`. This file is located in these directories:

UNIX:

```
$RTHOME/include/$RTARCH/rtworks
```

OpenVMS:

```
RTHOME: [ INCLUDE.RTWORKS ]
```

Windows:

```
%RTHOME%\include\rtworks
```

The SmartSockets IPC API includes all the functions used for interprocess communication.

Logical Connection Names

SmartSockets simplifies the creation of connections with logical connection names that are specified consistently for all protocols. A server process uses a logical connection name to create a server connection, and a client process uses the same logical connection name to create a client connection to the server process. Each logical connection name has the form:

protocol:node:address

which can be shortened to *protocol:node*, *protocol*, or simply *node* for normal connections. For the client process to find the server process, the logical connection name used by the client must exactly match the logical connection name used by the server (for example, the name `tcp:moe:1234` does not match the name `tcp:conan:1234`). The exception to this is when the server that the client process is trying to find has more than one IP address. For more information on this case, see Multiple IP Addresses on page 102.

This section describes the features of logical connection names in peer-to-peer connections. For a discussion of how RTserver and RTclient add more function to logical connection names, such as search lists and abbreviated names, see Logical Connection Names for RT Processes on page 192.

Protocol Portion

The *protocol* part of the connection name refers to an IPC protocol type. The valid values for *protocol* are `local` and `tcp`. RTclient and RTserver can also use the `udp_broadcast` protocol to find an RTserver. See The Udp_Broadcast Protocol on page 193 for more details on the `udp_broadcast` protocol.

Node Portion

The *node* part of the connection name refers to a computer node name. The special value `_node` can be used for *node* to indicate the name of the current node; this is useful for generalizing a connection name to work on any computer. See the `TutSocketCreate*` functions in the *TIBCO SmartSockets Utilities* reference for more details on the node name formats.

Address Portion

The *address* part of the connection name refers to a protocol-specific IPC location, such as a TCP port number. The addresses for all protocols are:

Protocol Name	Description of Address
local	file name in the directory specified by the function TutGetSocketDir
tcp	TCP port number or service name
udp_broadcast	UDP port number or service name

For the `tcp` and `udp_broadcast` protocols, the address can be either an integer port number or a service name accessible with the `getservbyname` socket function (internally SmartSockets always uses "`tcp`" for the *proto* second argument to `getservbyname`, even for `udp_broadcast` logical connection names).

Multiple IP Addresses

Generally, for the client process to find a server process, the logical connection name used by the client must exactly match the logical connection name used by the server. For example, the name `tcp:moe:1234` does not match the name `tcp:conan:1234`. However, this perfect match is not necessarily needed when the client is trying to find a server that has more than one IP address. A server or host with more than one IP address is called a multi-homed host. If the server uses `_any` as its node in the logical connection name, the server listens on all IP addresses. Using `_node` causes the server to listen only on the default IP address.



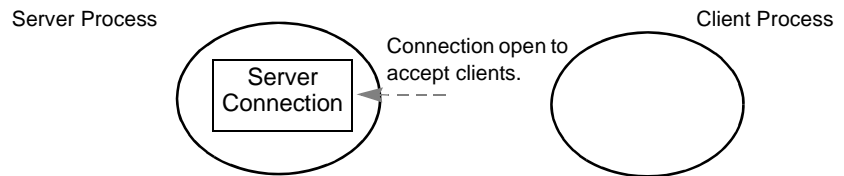
Using the keyword `_any` in the logical connection name is discouraged for server to server connections. When a server connects to another server whose logical connection name uses `_any`, the server might attempt to reconnect every `Server_Reconnect_Interval` seconds. This is a known problem and will be fixed in a future release.

Creating Connections

The standard client-server model for network applications is used when creating connections. The server starts first and waits to be contacted by the client. Once the server and client make contact, they become equal partners, and the model switches to a peer-to-peer model. Three steps (as shown in Figure 6, Figure 7, and Figure 8) are required when two processes create connections to each other.

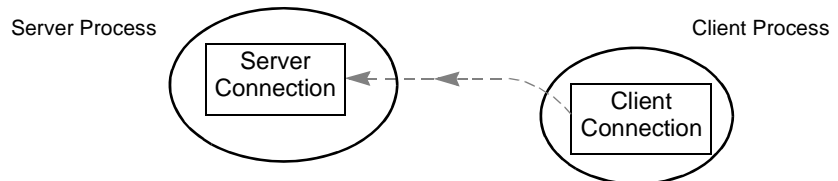
1. The server process creates a server connection that is used to accept client processes. After creating this connection, the server process waits for client processes to connect.

Figure 6 Server Creates a Connection



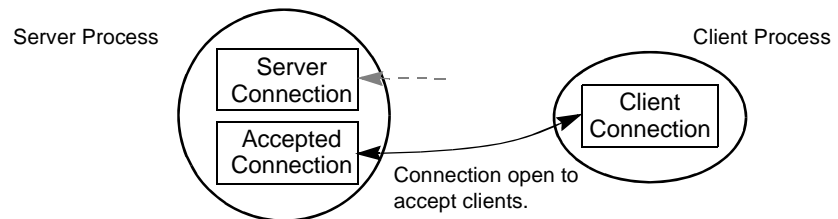
2. A client process creates a client connection and rendezvous with the server connection.

Figure 7 Client Creates Connection and Rendezvous With Server Connection



3. The server process accepts the client connection and creates a new connection for that client, thereby establishing a communication link, and keeps its original connection open for accepting connections from other client processes.

Figure 8 Server and Client Connection



The concepts of servers and clients only apply when the two processes are creating connections. Once the server has accepted the client, both processes have equivalent connections to each other. Both processes can send and receive messages using their connections, and there is no conceptual difference between the server and the client process. The model switches from client-server to peer-to-peer. A single process can have any number of server connections, client connections, and connections accepted from clients.

Creating a Server Connection

The server connection, which can use any one of the supported protocols, is created by calling the function `TipcConnCreateServer` with a logical connection name parameter used to identify the server connection. For example:

```
server_conn = TipcConnCreateServer("tcp:_node:5252");
if (server_conn == NULL) {
    TutOut("Could not create server connection: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

The only purpose of a server connection is to accept client connections. Messages cannot be sent or received on a server connection. Two server connections on the same node using the same IPC protocol cannot use the same logical connection name address portion.

Creating a Client Connection

In a fashion similar to server connections, a client connection is created by calling the function `TipcConnCreateClient` with a logical connection name parameter that must match the name used with `TipcConnCreateServer`. For example:

```
conn = TipcConnCreateClient("tcp:_node:5252");
if (conn == NULL) {
    TutOut("Could not create connection to server: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

The client connection must use the same IPC protocol as the server. The client connection must be created after the server connection, as the client needs something to contact.

The Server Accepts the Client

Once the client has called `TipcConnCreateClient` to create its connection, the server process must accept the client by calling the function `TipcConnAccept`. For example:

```
client_conn = TipcConnAccept(server_conn);
if (client_conn == NULL) {
    TutOut("Could not accept client: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

When the server process accepts the client, a new connection is created and returned by `TipcConnAccept`. This new connection is the peer-to-peer link for the server process to communicate with the client process. The client process gets its peer-to-peer link directly from `TipcConnCreateClient`. After each call to `TipcConnAccept`, the original server connection is unchanged and can be used to accept more clients later.

A server process can accept any number of clients by calling `TipcConnAccept` once for each client. Each connection does use an operating system file descriptor, though, and most operating systems have a limit on how many open file descriptors a process can have. See [File Descriptor Upper Limit](#) on page 148 for information on how to increase the open file descriptor limit.

While `TipcConnCreateServer` can execute and return immediately, both `TipcConnCreateClient` and `TipcConnAccept` do not return until both sides of the peer-to-peer link are established. During the transition from client-server to peer-to-peer, the client and server simultaneously exchange some initial handshaking information, such as the SmartSockets IPC protocol version (which is defined in `<rtworks/ipc.h>` as `T_IPC_PROTOCOL_VERSION`), integer number format (C type `T_INT_FORMAT`), and real number format (C type `T_REAL_FORMAT`). One process cannot be both the server and client ends of the same connection due to this simultaneous handshaking. The protocol version exchange prevents incompatible clients and servers from contacting each other, and also allows newer protocol versions to communicate with older protocol versions. Future versions of SmartSockets will be protocol compatible with the current version, and new message and connection features will be added in a compatible way to allow old and new versions to interoperate. The integer and real number formats are used to convert fields in messages when messages are sent between heterogeneous nodes (see [Sending Messages in a Heterogeneous Environment](#) on page 123 for information on the handling of messages in a heterogeneous environment).

If there are no clients waiting to be accepted, `TipcConnAccept` does not return until a client does connect (unless non-blocking operations are enabled with `TipcConnSetBlockMode`). The server process can use the function `TipcConnCheck` to check if a client has connected:

```
if (TipcConnCheck(server_conn, T_IPC_SOCKET_CHECK_READ)) {
    TutOut("A client is waiting to be accepted.\n");
}
```

Destroying a Connection

When a process is finished with a connection, the connection should be destroyed to free up the memory it occupies and close the socket. This is done with the `TipcConnDestroy` function.

If a process no longer wants to accept clients on a server connection, that connection can be destroyed.

For example:

```
if (!TipcConnDestroy(server_conn)) {
    TutOut("Could not destroy server connection: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

If a client process is done with a connection, the connection can be destroyed. For example:

```
if (!TipcConnDestroy(conn)) {
    TutOut("Could not destroy connection: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

When a process or task exits, the operating system automatically closes any open sockets the process had. When a peer-to-peer socket of a connection is closed, the process at the other end eventually gets an error number of `T_ERR_EOF` when trying to read or write data on its connection.

A process with connections using the `tcp` protocol may lose outgoing messages if the process terminates without calling `TipcConnDestroy` to destroy each connection. TCP/IP's `SO_LINGER` option, which preserves data, is ignored when closing a socket that has non-blocking I/O enabled. While data loss rarely occurs on UNIX and OpenVMS, it can happen frequently on Windows.

`TipcConnDestroy` sets the block mode of the connection to `FALSE` before closing the connection's socket, which forces the operating system to deliver all flushed outgoing messages.

Callbacks

Once two processes have created connections to each other, each process can create connection callbacks to be executed when certain operations occur. Callbacks are conceptually equivalent to dynamically adding a line of code to a program. The *SmartSockets Utilities* reference contains more information on callbacks, which are manipulated using utilities.

Process Callbacks

Connection process callbacks are executed while processing a message with the function `TipcConnMsgProcess`. This callback type is the most frequently used. A process callback can be called for a specific type of message or created globally and called for all message types. For example, a process callback can be created for the `NUMERIC_DATA` message type. When any message of that type is processed by calling `TipcConnMsgProcess`, the process callback is called. If the process callback is created globally, it is called for all `NUMERIC_DATA` type messages as well as any other type of message.

Function to create callback:

`TipcConnProcessCbCreate`. For example:

```
if (TipcConnProcessCbCreate(conn, mt, my_func, my_arg) == NULL) {
    /* error */
}
```

Function to look up callback:

`TipcConnProcessCbLookup`. For example:

```
cb = TipcConnProcessCbLookup(conn, mt, my_func, my_arg);
if (cb == NULL) {
    /* error */
}
```

Default Callbacks

Connection default callbacks are executed while processing a message with the function `TipcConnMsgProcess` if a process callback specific to the message type (that is, not a global process callback) has not been called. Default callbacks are useful for processing unexpected message types or for generic processing of most message types. For example, you have a client process to manage and display alarm messages. The only process callback you would want for that client process would be for messages of type "alarm." You could then write a default callback to process any other type of data received and display a warning.

Function to create callback:

`TipcConnDefaultCbCreate`. For example:

```
if (TipcConnDefaultCbCreate(conn, my_func, my_arg) == NULL) {
    /* error */
}
```

Function to look up callback:

`TipcConnDefaultCbLookup`. For example:

```
cb = TipcConnDefaultCbLookup(conn, my_func, my_arg);
if (cb == NULL) {
    /* error */
}
```

Read Callbacks

Connection read callbacks are executed when an incoming message is read from a connection's socket into the read buffer and first unpacked into a message. Read callbacks are most commonly used for writing incoming messages to message files.

Function to create callback:

`TipcConnReadCbCreate`. For example:

```
if (TipcConnReadCbCreate(conn, mt, my_func, my_arg) == NULL) {
    /* error */
}
```

Function to look up callback:

TipcConnReadCbLookup. For example:

```
cb = TipcConnReadCbLookup(conn, mt, my_func, my_arg);
if (cb == NULL) {
    /* error */
}
```

Write Callbacks

Connection write callbacks are executed when an outgoing message is sent to a connection (that is, copied to a connection's write buffer). Write callbacks are most commonly used for writing outgoing messages to message files.

Function to create callback:

TipcConnWriteCbCreate. For example:

```
if (TipcConnWriteCbCreate(conn, mt, my_func, my_arg) == NULL) {
    /* error */
}
```

Function to look up callback:

TipcConnWriteCbLookup. For example:

```
cb = TipcConnWriteCbLookup(conn, mt, my_func, my_arg);
if (cb == NULL) {
    /* error */
}
```

Queue Callbacks

Connection queue callbacks are executed when a message is inserted into or deleted from a connection's message queue. Queue callbacks are useful for watching the messages that have been read in from a connection's socket and inserted into the message queue, but not yet processed.

Function to create callback:

TipcConnQueueCbCreate. For example:

```
if (TipcConnQueueCbCreate(conn, mt, my_func, my_arg) == NULL) {
    /* error */
}
```

Function to look up callback:

TipcConnQueueCbLookup. For example:

```
cb = TipcConnQueueCbLookup(conn, mt, my_func, my_arg);
if (cb == NULL) {
    /* error */
}
```

Error Callbacks

Connection error callbacks are executed when an unrecoverable error occurs. These errors include socket problems and network failures such as:

- A read timeout has occurred and the keep alive then failed.
- A write timeout has occurred.
- A read operation on the connection's socket failed. The most common cause of this error is that the process at the other end of the connection has destroyed its connection, which will close the socket.
- A write operation on the connection's socket failed. The most common cause of this error is that the process at the other end of the connection has destroyed its connection, which will close the socket.

Function to create callback:

TipcConnErrorCbCreate. For example:

```
if (TipcConnErrorCbCreate(conn, my_func, my_arg) == NULL) {
    /* error */
}
```

Function to look up callback:

TipcConnErrorCbLookup. For example:

```
cb = TipcConnErrorCbLookup(conn, my_func, my_arg);
if (cb == NULL) {
    /* error */
}
```


Example of Process Callback

This creates a process callback, called whenever a NUMERIC_DATA message is processed:

```
mt = TipcMtLookupByNum(T_MT_NUMERIC_DATA);
if (mt == NULL) {
    TutOut("Could not look up NUMERIC_DATA msg type: error
<%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (TipcConnProcessCbCreate(client_conn, mt,
cb_process_numeric_data,
                        NULL) == NULL) {
    TutOut("Could not create NUMERIC_DATA process cb: error
<%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

Example of Default Callback

This creates a default callback, called when a message is processed that has no process callbacks:

```
if (TipcConnDefaultCbCreate(client_conn,
                        cb_default, NULL) == NULL) {
    TutOut("Could not create default cb: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

Example of Read Callback

This example creates a global read callback that is called whenever any message is read from a connection:

```
if (TipcConnReadCbCreate(client_conn,
                        NULL, /* global callback */
                        cb_read,
                        msg_file) == NULL) {
    TutOut("Could not create read cb: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

Example of Write Callback

This example creates a global write callback that is called whenever any message is written to a connection:

```
if (TipcConnWriteCbCreate(conn,
                          NULL, /* global callback */
                          cb_write,
                          msg_file) == NULL) {
    TutOut("Could not create write cb: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

Example of Queue Callback

This example creates a global queue callback that is called whenever any message is inserted into or deleted from a connection's message queue:

```
if (TipcConnQueueCbCreate(client_conn,
                          NULL, /* global callback */
                          cb_queue,
                          NULL) == NULL) {
    TutOut("Could not create queue cb: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

Example of Error Callback

This example creates an error callback that is called whenever a non-recoverable error occurs on a connection:

```
if (TipcConnErrorCbCreate(client_conn,
                          cb_error,
                          NULL) == NULL) {
    TutOut("Could not create error cb: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

Callback Functions

Each connection callback has a specific function (such as `cb_process_numeric_data`) and argument (such as `NULL`). Several types of connection callbacks, such as process callbacks, also have a message type (such as `NUMERIC_DATA`) associated with them. The most commonly used connection callback type is the process callback. Process callback functions are used to perform the main processing of a message.

The following section describes a callback function in detail. This callback function, which is called when a message of type `NUMERIC_DATA` is processed with `TipConnMsgProcess`, simply accesses and prints the fields of the message:

```
/*..cb_process_numeric_data -- process callback for NUMERIC_DATA */
static void T_ENTRY cb_process_numeric_data(
    T_IPC_CONN conn,
    T_IPC_CONN_PROCESS_CB_DATA data,
    T_CB_ARG arg )
```

Notice that this callback is declared with the macro `T_ENTRY`. This is necessary for cross-platform portability. All prototypes and definitions of callbacks and thread functions must be declared with `T_ENTRY`.

All SmartSockets callback functions do not return a value and take three parameters:

- an opaque data type (type `T_IPC_CONN` in the case of connection callbacks)
- a non-opaque callback type-specific data argument
- a user-defined argument

```
{
    T_STR name;
    T_REAL8 value;

    /* set current field to first field in message */
    if (!TipMsgSetCurrent(data->msg, 0)) {
        TutOut("Could not set current field of message: error <%s>.\n",
            TutErrStrGet());
        return;
    }
}
```

The message being processed is available in the `data` parameter. Every callback function should first set the current field of the message before accessing the message fields, as the current field could be set to any field when the callback function is called.

```
/* access and print fields */
while (TipMsgNextStrReal8(data->msg, &name, &value)) {
    TutOut("%s = %s\n", name, TutRealToStr(value));
}
```

```
/* make sure we reached the end of the message */
if (TutErrNumGet() != T_ERR_MSG_EOM) {
    TutOut("Did not reach end of message: error <%s>.\n",
        TutErrStrGet());
}
} /* cb_process_numeric_data */
```

The above while loop accesses and prints all the name-value pairs from the NUMERIC_DATA message. The final error check for T_ERR_MSG_EOM ensures that the while loop reached the end of the message as expected (that is, some other error condition such as a message field type mismatch did not occur). Each application can add more or less error checking as desired.

Type-Specific Data Arguments

The non-opaque callback type-specific data argument varies among callback types. All SmartSockets callback type-specific data argument types must have a T_CB callback value as their first field.

Table 4 shows the callback type-specific data argument types for all connection callback types and the fields in those argument types.

Table 4 Connection Callback Types

Callback Type	Type of Second Parameter to Callback Functions	Fields in This Type
process	T_IPC_CONN_PROCESS_CB_DATA	T_CB <i>cb</i> ; T_IPC_MSG <i>msg</i> ;
default	T_IPC_CONN_DEFAULT_CB_DATA	T_CB <i>cb</i> ; T_IPC_MSG <i>msg</i> ;
read	T_IPC_CONN_READ_CB_DATA	T_CB <i>cb</i> ; T_IPC_MSG <i>msg</i> ; T_INT4 <i>packet_size</i> ;
write	T_IPC_CONN_WRITE_CB_DATA	T_CB <i>cb</i> ; T_IPC_MSG <i>msg</i> ;
queue	T_IPC_CONN_QUEUE_CB_DATA	T_CB <i>cb</i> ; T_IPC_MSG <i>msg</i> ; T_BOOL <i>insert_flag</i> ; T_INT4 <i>position</i> ;

Table 4 Connection Callback Types (Cont'd)

Callback Type	Type of Second Parameter to Callback Functions	Fields in This Type
error	T_IPC_CONN_ERROR_CB_DATA	T_CB <i>cb</i> ; T_INT4 <i>err_num</i> ; T_INT4 <i>c_err_num</i> ; T_INT4 <i>os_err_num</i> ; T_INT4 <i>socket_err_num</i> ;

Receiving and Processing Messages

As described earlier, a connection has a priority queue of incoming messages. The message at the front of this queue can be retrieved with the function `TipcConnMsgNext`. This message is normally processed immediately with the function `TipcConnMsgProcess` and then destroyed with the function `TipcMsgDestroy`.

For example:

```
while ((msg = TipcConnMsgNext(client_conn, T_TIMEOUT_FOREVER))
      != NULL) {
    if (!TipcConnMsgProcess(client_conn, msg)) {
        TutOut("Could not process message: error <%s>.\n",
              TutErrStrGet());
    }
    if (!TipcMsgDestroy(msg)) {
        TutOut("Could not destroy message: error <%s>.\n",
              TutErrStrGet());
    }
}

/* make sure we reached the end of the data */
if (TutErrNumGet() != T_ERR_EOF) {
    TutOut("Did not reach end of data: error <%s>.\n",
          TutErrStrGet());
}
```

The above while loop receives and processes all messages from the connection, and the final error check for `T_ERR_EOF` ensures that the while loop reached the end of the data as expected. Each application can add more or less error checking as desired.

If there are no messages in the connection's message queue, then `TipcConnMsgNext` reads data from the connection by calling the function `TipcConnRead` over and over until a full message has been read in, a certain period of time has elapsed, or an error occurs.

As `TipcConnRead` reads in each message packet, it converts the message packet into a message. The connection read callbacks are executed by `TipcConnRead` each time a full message is read into the connection's read buffer. The connection queue callbacks are executed by `TipcConnMsgInsert` each time a message is inserted into a connection's message queue, and by `TipcConnMsgNext` and `TipcConnMsgSearch` each time a message is removed from a connection's message queue.

If necessary, `TipcConnRead` converts the integers, real numbers, and strings in the message header to the format used by the receiving process (see [Sending Messages in a Heterogeneous Environment](#) on page 123 to learn more about sending messages between different types of computers). `TipcConnRead` also performs some checks for duplicate messages if the received message was sent with GMD. See [Receiving Messages](#) on page 337 for details on the GMD-specific aspects of receiving messages.

The second parameter to `TipcConnMsgNext` specifies the maximum amount of time (in seconds) that it waits for an incoming message to arrive. The timeout `0.0` can be used to get the next message that is immediately available, the timeout `T_TIMEOUT_FOREVER` can be used to wait indefinitely for the next message, and any non-negative timeout can be used to wait for a certain period of time for the next message.

`TipcConnMsgProcess` executes the connection default callbacks only if there are no connection process callbacks for the type of the message being processed. The two levels of callbacks allow for greater flexibility: process callbacks can handle message type-specific needs and default callbacks can be used to handle generic needs. While most messages should be processed with `TipcConnMsgProcess` so that the connection process and default callbacks can be executed, there is no requirement that all messages be processed this way.

If a process does not need the flexibility and extensibility that connection callbacks offer, it can call `TipcConnMsgNext` to get a message and then access the fields of the message directly with the `TipcMsgNextType` functions. All of the `SmartSockets` modules such as `RTdaq`, however, use callbacks to process messages, so that user-defined callbacks can also be added to extend the function.

Using the TipcConnMainLoop Convenience Function

The TipcConnMainLoop convenience function receives and processes messages by calling TipcConnMsgNext, TipcConnMsgProcess, and TipcMsgDestroy over and over. Using this function, the previous loop is rewritten as shown:

```
if (!TipcConnMainLoop(client_conn, T_TIMEOUT_FOREVER)) {
    /* make sure we reached the end of the data */
    if (TutErrNumGet() != T_ERR_EOF) {
        TutOut("Did not reach end of data: error <%s>.\n",
              TutErrStrGet());
    }
}
```

Searching for Incoming Messages With the TipcConnMsgSearch Function

In most situations, incoming messages should be accessed sequentially with TipcConnMsgNext. Sometimes, though, messages need to be accessed in non-priority order. Examples of this include remote procedure calls (see Remote Procedure Calls on page 147 for more information) and checking for a message of a specific type. The function TipcConnMsgSearch is used to search for a specific message. For example, to search for a NUMERIC_DATA message, this search function could be used:

```
/* ===== */
/*..search_numeric_data -- search for NUMERIC_DATA */
static T_BOOL T_ENTRY search_numeric_data(conn, msg, arg)
T_IPC_CONN conn;
T_IPC_MSG msg;
T_PTR arg;
{
    T_BOOL status;
    T_IPC_MT mt;
    T_INT4 num;

    if (!TipcMsgGetType(msg, &mt)) {
        TutOut("could not get msg type: error <%s>\n",
              TutErrStrGet());
        (void)TipcMsgPrint(msg, TutOut);
        return FALSE;
    }
    if (!TipcMtGetNum(mt, &num)) {
        TutOut("could not get message type num: error <%s>\n",
              TutErrStrGet());
        (void)TipcMsgPrint(msg, TutOut);
        return FALSE;
    }
}
```

```

    /* return TRUE if message type matches what we want */
    return num == T_MT_NUMERIC_DATA;
} /* search_numeric_data */

```

The above search function `search_numeric_data` could then be used as follows:

```

msg = TipcConnMsgSearch(conn, T_TIMEOUT_FOREVER,
                        search_numeric_data, NULL);
if (msg == NULL) {
    TutOut("Could not find NUMERIC_DATA message: error <%s>.\n",
          TutErrStrGet());
}

```

`TipcConnMsgSearch` traverses a connection's message queue and calls a user-defined search function once for each message. If the end of the message queue is reached, then `TipcConnMsgSearch` calls the function `TipcConnRead` to receive more messages into the message queue and then starts over again at the front of the queue. `TipcConnMsgSearch` returns when, the search function returns `TRUE`, a certain period of time has elapsed, or an error occurs.

If the search function returns `TRUE`, then `TipcConnMsgSearch` removes the desired message from the connection's message queue and returns the message. The second parameter to `TipcConnMsgSearch` specifies a timeout just like the second parameter to `TipcConnMsgNext`.

Using the `TipcConnMsgSearchType` Convenience Function

The `TipcConnMsgSearchType` convenience function searches for a message with a specific type by calling `TipcConnMsgSearch`. Using this function, the code is rewritten as shown in the example:

```

mt = TipcMtLookupByNum(T_MT_NUMERIC_DATA);
if (mt == NULL) {
    TutOut("Could not look up NUMERIC_DATA msg type: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

msg = TipcConnMsgSearchType(conn, T_TIMEOUT_FOREVER, mt);
if (msg == NULL) {
    TutOut("Could not find NUMERIC_DATA message: error <%s>.\n",
          TutErrStrGet());
}

```

Note that if `TipcConnMsgSearchType` is used, no search function is needed.

Buffering of Incoming Messages

When a message is received through a connection, it passes through several buffers before finally being accessed with `TipcConnMsgNext` or `TipcConnMsgSearch`. A buffer is an area of memory where data, such as messages, is stored while waiting to be accessed or transmitted. For incoming messages, these buffers are used:

- the connection's socket's incoming buffer
- the connection's read buffer
- the connection's message queue

Most of these buffers cannot be accessed directly, but inefficient use of these buffers can reduce the performance of connections. When messages are transferred from the connection's socket's incoming buffer to the connection's read buffer (this is done by `TipcConnRead`), an (operating system) system call is performed. System call functions are necessary, but they are much more time-consuming than normal functions, and better performance can be achieved by batching data transfers into fewer system calls. A connection's read buffer helps to batch incoming data transfers by allowing `TipcConnRead` to read as much data as possible for each system call.

Because sockets are byte streams, large messages may be received in pieces, which are buffered in the connection's read buffer until the entire message is received. Once the full incoming message is received, it is transferred to the connection's message queue and is available to be accessed and processed.

Sending Messages

Once a message has been constructed with the `TipcMsg*` functions, it can be sent through the connection with the function `TipcConnMsgSend`. The code constructs and sends a `NUMERIC_DATA` message:

```
mt = TipcMtLookupByNum(T_MT_NUMERIC_DATA);
if (mt == NULL) {
    TutOut("Could not look up NUMERIC_DATA msg type: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

msg = TipcMsgCreate(mt);
if (msg == NULL) {
    TutOut("Could not create NUMERIC_DATA message: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

if (!TipcMsgWrite(msg,
                  T_IPC_FT_STR, "voltage",
                  T_IPC_FT_REAL8, 33.4534,
                  T_IPC_FT_STR, "switch_pos",
                  T_IPC_FT_REAL8, 0.0,
                  NULL)) {
    TutOut("Could not append to NUMERIC_DATA msg: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

if (!TipcConnMsgSend(conn, msg)) {
    TutOut("Could not send NUMERIC_DATA message: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

`TipcConnMsgSend` executes the connection write callbacks, converts the message into a message packet, and then appends the message packet to the end of the connection's write buffer. If the message's header string encode property is `TRUE`, the message header string properties, such as destination, are stored as four-byte integers in the message packet, which compresses the message header so as to use less network bandwidth. See [Header String Encode](#) on page 14 for more information about message header string encoding.

`TipcConnMsgSend` also saves a copy of the outgoing message in the connection GMD area if the message is sent with GMD. See [Sending Messages](#) on page 336 for details on the GMD-specific aspects of sending messages. `TipcConnMsgSend` then calls `TipcConnFlush` to flush the buffered outgoing data to the connection's socket if the connection's auto flush size is not `T_IPC_NO_AUTO_FLUSH`, and the number of bytes buffered in the connection's write buffer is larger than the value of the connection's auto flush size.

See Buffering of Outgoing Messages on page 122 for more information on how outgoing messages are buffered.

Using the TipConnMsgWrite Convenience Function

The TipConnMsgWrite convenience function handles a variable number of arguments and allows you to create a message, append fields, send the message on a connection, and destroy the message. The function takes enumerated values that begin with T_IPC_FT_TYPE, where *TYPE* is replaced by a field type as shown in Table 1 on page 7, such as T_IPC_FT_STR. A final parameter of NULL is used to terminate the variable number of arguments. Using this function, the above NUMERIC_DATA message could be constructed and sent as follows:

```
mt = TipMtLookupByNum(T_MT_NUMERIC_DATA);
if (mt == NULL) {
    TutOut("Could not look up NUMERIC_DATA msg type: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

if (!TipConnMsgWrite(conn, mt,
                    T_IPC_FT_STR, "voltage",
                    T_IPC_FT_REAL8, 33.4534,
                    T_IPC_FT_STR, "switch_pos",
                    T_IPC_FT_REAL8, 0.0,
                    NULL)) {
    TutOut("Could not append to NUMERIC_DATA message: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```



C/C++ functions that use a variable number of arguments often require fewer lines of code to use, but no type checking is done by the C/C++ compiler on the variable arguments.

As an added convenience, `TipcConnMsgWrite` also allows enumerated values that begin with `T_IPC_PROP_NAME`, where *NAME* is replaced by a message property name, such as `T_IPC_PROP_DELIVERY_MODE`. For example, a complex message can be constructed and sent with one call to `TipcConnMsgWrite`:

```
if (!TipcConnMsgWrite(conn, mt,
    T_IPC_PROP_SENDER, "/_conan_5415",
    T_IPC_PROP_DEST, "/system/thermal",
    T_IPC_PROP_PRIORITY, 2,
    T_IPC_PROP_DELIVERY_MODE, T_IPC_DELIVERY_ALL,
    T_IPC_PROP_DELIVERY_TIMEOUT, 20.0,
    T_IPC_PROP_LB_MODE, T_IPC_LB_WEIGHTED,
    T_IPC_PROP_HEADER_STR_ENCODE, TRUE,
    T_IPC_PROP_USER_PROP, 42,
    T_IPC_FT_STR, "voltage",
    T_IPC_FT_REAL8, 33.4534,
    T_IPC_FT_STR, "switch_pos",
    T_IPC_FT_REAL8, 0.0,
    NULL)) {
    /* error */
}
```

Buffering of Outgoing Messages

When a message is sent through a connection by calling `TipcConnMsgSend`, it passes through several buffers. A buffer is an area of memory where data, such as messages, is stored while waiting to be accessed or transmitted. For outgoing messages, these buffers are used:

- the connection's write buffer
- the connection's socket's outgoing buffer

These buffers cannot be accessed directly, but inefficient use of these buffers can reduce the performance of connections. When messages are transferred from the connection's write buffer to the connection's socket's outgoing buffer (this is done by `TipcConnFlush`), an (operating system) system call is performed. System call functions are necessary, but they are much more time-consuming than normal functions, and better performance can be achieved by batching data transfers into fewer system calls. A connection's write buffer helps to batch outgoing data transfers by allowing `TipcConnFlush` to write as much data as possible for each system call.

In most situations, the connection's auto flush size property can be used to control when the function `TipcConnFlush` is called automatically. Sometimes, though, a program does need to use `TipcConnFlush` explicitly to force outgoing buffered messages to be flushed. If a program sends messages and then does not call any other `TipcConn*` functions for a period of time, the outgoing messages sit in the connection's write buffer. If the program sends messages and then calls `TipcConnMsgNext` right away to get the next available message, the program

does not need to call `TipcConnFlush` (as `TipcConnMsgNext` calls `TipcConnRead`, which automatically flushes the buffered outgoing data). Flushing the write buffer before reading allows responses to the outgoing messages to be available sooner.

Sending Messages in a Heterogeneous Environment

SmartSockets allows messages to be sent between processes on different types of computers, such as Intel x86 and Sun SPARC. Converting a message from one kind of platform to another is handled transparently by the SmartSockets API. The rest of this section describes in detail how this is done.

In a heterogeneous environment, different computer nodes on a network often have incompatible integer, real number, and string formats. A common example is Sun SPARC and Intel x86. All of the platforms supported by SmartSockets use two's complement integer arithmetic, but there are two common integer layouts:

- big endian — the least significant byte of a number is in the highest address
- little endian — the least significant byte of a number is in the lowest address

Most non-Digital and non-Intel computers use big-endian format. Converting integers between big-endian and little-endian formats is simple and involves byte-swapping the entire integer. Only two-byte (C type `T_INT2`), four-byte integers (C type `T_INT4`), and eight-byte integers (C type `T_INT8`) have to be byte-swapped; one-byte (C type `T_CHAR`) integers and `NULL`-terminated character strings do not need any conversion when moving data between platforms using the single byte, US ASCII character set. Additional character set considerations are discussed later in this section.

A similar situation exists for real numbers. These real number formats exist for the platforms supported by SmartSockets:

- IEEE — almost-universal floating-point format
- DEC D — default format for OpenVMS VAX 8-byte real numbers (C type `T_REAL8`)
- DEC F — format for OpenVMS 4-byte real numbers (C type `T_REAL4`)
- DEC G — default format for OpenVMS AXP 8-byte real numbers (C type `T_REAL8`)

Most platforms supported by SmartSockets use IEEE floating-point format for real numbers. In addition to the above formats, real numbers are subject to the same endianness byte order that integers use. For example, IEEE numbers are big-endian on most platforms but are little-endian on a few platforms.

For characters and character strings, there are two possibilities:

- ASCII — ANSI standard character format
- EBCDIC — MVS character format

When a message is read from a connection, the integers, real numbers, and strings within the message header are automatically converted from the formats of the sending process to the formats of the receiving process by the function `TipcConnRead`. The message data information is converted on an incremental per-field basis by the `TipcMsgNext*` functions. This conversion scheme is commonly known as receiver-makes-right data conversion. Because the data fields in a message are only converted when the data is accessed, RTserver routes publish-subscribe messages without performing any conversion on the data fields.

Message conversion always takes place in the receiving process and only happens when necessary (that is, no conversion happens when both processes have the same formats). As described in *The Server Accepts the Client* on page 105, when two processes create connections to each other, they first exchange with each other their integer format and real number format (the character format is deduced from the real number format). This format information is used by `TipcConnRead` to determine when conversion is needed.

Using Threads With Connections

Threads are a widely supported abstraction for concurrent programming. Many modern operating systems have incorporated threads to leverage the availability of relatively low-cost multiprocessor hardware. In fact, this often represents the compelling reason to incorporate threads into any software design. A concurrent, multithreaded program has the potential to fully utilize hardware platforms with two or more processors, whereas a sequential, single-threaded program does not. Even on uniprocessor systems, multithreaded programs can improve throughput by overlapping processing and I/O requests without resorting to relatively complex and non-portable asynchronous I/O facilities. Threads have also long been recognized as a tool for implementing demanding software requirements for high availability and responsiveness. SmartSockets is designed to support the multiple-threads for each connection model and the single-thread for each connection model of multithreaded applications.

The benefits of a multithreaded application, however, must be weighed carefully against its costs in resource consumption, efficiency, and program complexity. Because threads operate within a shared address space, their activities must be carefully synchronized to insure that they do not interfere with one another. If an application does not lend itself to partitioning into discrete units of processing of a reasonable size, synchronization overhead can quickly outweigh the benefits of concurrency even on multiprocessor hardware platforms. Threads are simply not appropriate for all programs.

It is worth noting, however, that SmartSockets applications often do lend themselves well to multithreaded implementations. This is particularly true of server processes, which may get multiple requests from independent clients simultaneously. These requests must be handled serially if the server has only one thread of control, allowing long requests to block the servicing of other pending requests. Multithreaded processes can be more adaptive to such variations, allowing short requests to complete out of sequence. The use of multiple threads also offers an alternative to the UNIX convention of forking a child process to deal with each new client.

SmartSockets connections are designed to serve as high-level synchronization objects in multithreaded applications. Each connection has a set of mutexes (mutual exclusion locks) that are used to ensure that threads sharing the same connection do not interfere with each other, yet can operate concurrently where there is no chance of interference. For instance, one thread may be sending a message on a connection, while a second thread is retrieving the next received message from the connection's queue, and a third and fourth thread are both running one of the connection's process callbacks.

The following server and client examples demonstrate how to write a connection-based server program that handles multiple simultaneous clients by using a separate thread for each client connection. To simplify the example, a fixed number of server threads (controlled by `#define SERVER_COUNT 2`) are created in advance. This determines the maximum number of client connections that may be simultaneously serviced. Should more than `SERVER_COUNT` clients attempt to connect to the server, excess clients would be unable to successfully complete their connection until one of the server threads becomes available.

There is also an example that extends this server to use multiple threads for each client connection. This allows multiple messages from the same client to be processed in parallel instead of serially, thus substantially improving response times. See [Adding Multiple Threads for a Client](#) on page 134 for details.

The source code files for this example are located in these directories:

UNIX:

`$RTHOME/examples/smrtsock/manual`

OpenVMS:

`RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]`

Windows:

`%RTHOME%\examples\smrtsock\manual`

The online source files have additional `#ifdefs` to provide C++ support. These `#ifdefs` are not shown to simplify the example.

Example 9 Multithreaded Server Source Code

/ connmts1.c -- multithreaded connections example server 1 */*

*/**

This program uses multiple server threads which allows multiple clients to connect and submit messages simultaneously.

**/*

```
#include <rtworks/ipc.h>
```

```
#define SERVER_COUNT 2
```

```
T_IPC_CONN server_conn;
```

```
T_TSD_KEY id_key = T_INVALID_TSD_KEY;
```



```

/* ===== */
/*..cb_process_numeric_data -- process callback for NUMERIC_DATA */
void T_ENTRY cb_process_numeric_data(
    T_IPC_CONN conn,
    T_IPC_CONN_PROCESS_CB_DATA data,
    T_CB_ARG arg )
{
    T_STR id_str;
    T_INT4 i;
    T_STR var_name[3];
    T_REAL8 var_value[3];
    T_REAL8 in_time;
    T_REAL8 out_time;
    T_STR the_time = NULL;

    if (!TutTsdGetValue(id_key, &id_str)) {
        TutOut("Could not get TSD value for thread: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    /* set current field to first field in message */
    if (!TipcMsgSetCurrent(data->msg, 0)) {
        TutOut("%s: Could not set current field of msg: error <%s>.\n",
            id_str, TutErrStrGet());
        TutThreadExit(NULL);
    }

    for (i = 0; i < 3; i++) {
        if (!TipcMsgNextStrReal8(data->msg,
            &var_name[i], &var_value[i])) {
            TutOut("%s: Could not parse NUMERIC_DATA msg: error <%s>.\n",
                id_str, TutErrStrGet());
            TutThreadExit(NULL);
        }
    }

    in_time = TutGetWallTime();
    the_time = TutStrDup(TutTimeNumToStr(in_time));

    TutOut("%s: (%s %d %s %s %s %s) at %s\n", id_str,
        var_name[0], (T_INT4)var_value[0],
        var_name[1], TutRealToStr(var_value[1]),
        var_name[2], TutTimeNumToStr(var_value[2]),
        the_time);

    TutFree(the_time);
    TutSleep(var_value[1]);

    out_time = TutGetWallTime();

```

```

        if (!TipcMsgWrite(data->msg,
                        T_IPC_FT_STR, "in-time",
                        T_IPC_FT_REAL8, in_time,
                        T_IPC_FT_STR, "out-time",
                        T_IPC_FT_REAL8, out_time,
                        NULL)) {
            TutOut("%s: Could not append to NUMERIC_DATA msg: error
<%s>.\n",
                id_str, TutErrStrGet());
            TutThreadExit(NULL);
        }

        if (!TipcConnMsgSend(conn, data->msg)) {
            TutOut("%s: Could not send NUMERIC_DATA msg: error <%s>.\n",
                id_str, TutErrStrGet());
            TutThreadExit(NULL);
        }

        if (!TipcConnFlush(conn)) {
            TutOut("%s: Could not flush conn to client: error <%s>.\n",
                id_str, TutErrStrGet());
            TutThreadExit(NULL);
        }
    } /* cb_process_numeric_data */

/* ===== */
/*..server_thread -- thread function for server threads */
T_PTR T_ENTRY server_thread(arg)
T_PTR arg;
{
    T_STRING id_str;
    T_IPC_MT mt;
    T_IPC_CONN client_conn;
    T_INT4 i;

    sprintf(id_str, "%d", *(T_INT4 *)arg);
    if (!TutTsdSetValue(id_key, id_str)) {
        TutOut("Could not set TSD value for thread: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    mt = TipcMtLookupByNum(T_MT_NUMERIC_DATA);
    if (NULL == mt) {
        TutOut("%s: Could not look up NUMERIC_DATA mt: error <%s>.\n",
            id_str, TutErrStrGet());
        TutThreadExit(NULL);
    }

    for (i = 0; i < 2; i++) {
        client_conn = TipcConnAccept(server_conn);
        if (NULL == client_conn) {
            TutOut("%s: Could not accept client: error <%s>.\n",
                id_str, TutErrStrGet());
            TutThreadExit(NULL);
        }
    }
}

```

```

    TutOut("%s: Accepted client connection %d.\n", id_str, i + 1);
    if (TipcConnProcessCbCreate(client_conn, mt,
                               cb_process_numeric_data,
                               arg) == NULL) {
        TutOut("%s: Could not create NUMERIC_DATA process cb.\n",
               id_str);
        TutOut("  error <%s>.\n", TutErrStrGet());
        TutThreadExit(NULL);
    }

    if (!TipcConnMainLoop(client_conn, T_TIMEOUT_FOREVER)) {
        /* make sure we reached the end of the data */
        if (TutErrNumGet() != T_ERR_EOF) {
            TutOut("%s: Did not reach end of data: error <%s>.\n",
                   id_str, TutErrStrGet());
        }
    }

    TutOut("%s: Destroying client connection %d.\n", id_str, i +
1);
    if (!TipcConnDestroy(client_conn)) {
        TutOut("%s: Could not destroy client conn: error <%s>.\n",
               id_str, TutErrStrGet());
    }
}

return NULL;
} /* server_thread */

/* ===== */
/*..main -- main program */
int main()
{
    T_THREAD thread[SERVER_COUNT];
    T_INT4 id_args[SERVER_COUNT];
    T_INT4 i;

    if (!TipcInitThreads()) {
        TutOut("This platform does not support threads: error <%s>.\n",
               TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    if (!TutTsdKeyCreate(&id_key, 0)) {
        TutOut("Could not create TSD key: error <%s>.\n",
               TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    TutOut("Creating server connection.\n");
    server_conn = TipcConnCreateServer("tcp:_node:4000");
    if (NULL == server_conn) {
        TutOut("Could not create server connection: error <%s>.\n",
               TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
}

```

```

for (i = 0; i < SERVER_COUNT; i++) {
    id_args[i] = i + 1;
    thread[i] = TutThreadCreate((T_THREAD_FUNC)server_thread,
                                &id_args[i], NULL);
    if (TutThreadEqual(T_INVALID_THREAD, thread[i])) {
        TutOut("Could not create thread %d: error <%s>.\n",
               i, TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
}

for (i = 0; i < SERVER_COUNT; i++) {
    if (!TutThreadWait(thread[i], NULL)) {
        TutOut("Could not wait for thread %d: error <%s>.\n",
               i, TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
}

TutOut("Destroying server connection.\n");
if (!TipcConnDestroy(server_conn)) {
    TutOut("Could not destroy server connection: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

return T_EXIT_SUCCESS;
} /* main */

```

Example 10 Client Source Code

```
/* connmtc.c -- multithreaded connections example client */
```

```
/*
```

This program connects to the server process, submits several messages at once, and then outputs their individual response times.

```
*/
```

```
#include <rtworks/ipc.h>

#define T_NUM_JOB_TIMES 5
T_REAL8 job_times[] = { 1.0, 2.0, 7.0, 2.0, 1.0 };
/* ===== */
/*..cb_process_numeric_data -- process callback for NUMERIC_DATA */
void T_ENTRY cb_process_numeric_data(
    T_IPC_CONN conn,
    T_IPC_CONN_PROCESS_CB_DATA data,
    T_CB_ARG arg )
{
    T_INT4 i;
    T_STR var_name[5];
    T_REAL8 var_value[5];

    /* set current field to first field in message */
    if (!TipcMsgSetCurrent(data->msg, 0)) {
        TutOut("Could not set current field of message: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    for (i = 0; i < 5; i++) {
        if (!TipcMsgNextStrReal8(data->msg,
            &var_name[i], &var_value[i])) {
            TutOut("Could not parse NUMERIC_DATA message: error <%s>.\n",
                TutErrStrGet());
            TutExit(T_EXIT_FAILURE);
        }
    }

    TutOut("Reply (%s %d %s %s %s %s\n",
        var_name[0], (T_INT4)var_value[0],
        var_name[1], TutRealToStr(var_value[1]),
        var_name[2], TutTimeNumToStr(var_value[2]));
    TutOut("    %s %s ",
        var_name[3], TutTimeNumToStr(var_value[3]));
    TutOut("%s %s) elapsed %s seconds\n",
        var_name[4], TutTimeNumToStr(var_value[4]),
        TutRealToStr(var_value[4] - var_value[2]));
} /* cb_process_numeric_data */
```

```

/* ===== */
/*..main -- main program */
int main()
{
    T_IPC_CONN conn;
    T_IPC_MT mt;
    T_IPC_MSG msg;
    T_INT4 i;

    TutOut("Creating connection to server process.\n");
    conn = TipcConnCreateClient("tcp:_node:4000");
    if (NULL == conn) {
        TutOut("Could not connect to server process: error <%s>.\n",
TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    mt = TipcMtLookupByNum(T_MT_NUMERIC_DATA);
    if (NULL == mt) {
        TutOut("Could not look up NUMERIC_DATA msg type: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    if (TipcConnProcessCbCreate(conn, mt,
                                cb_process_numeric_data,
                                NULL) == NULL) {
        TutOut("Could not create NUMERIC_DATA process cb: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
    msg = TipcMsgCreate(mt);
    if (NULL == msg) {
        TutOut("Could not create NUMERIC_DATA message: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    for (i = 0; i < T_NUM_JOB_TIMES; i++) {
        if (!TipcMsgWrite(msg,
            T_IPC_FT_STR, "message-no",
            T_IPC_FT_REAL8, (T_REAL8)i + 1.0,
            T_IPC_FT_STR, "job-time",
            T_IPC_FT_REAL8, job_times[i],
            T_IPC_FT_STR, "submit-time",
            T_IPC_FT_REAL8, TutGetWallTime(),
            NULL)) {
            TutOut("Could not append to NUMERIC_DATA msg: error <%s>.\n",
                TutErrStrGet());
            TutExit(T_EXIT_FAILURE);
        }
    }
}

```

```

    if (!TipcConnMsgSend(conn, msg)) {
        TutOut("Could not send NUMERIC_DATA message: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    if (!TipcMsgSetNumFields(msg, 0)) {
        TutOut("Could not truncate NUMERIC_DATA message: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
}

if (!TipcMsgDestroy(msg)) {
    TutOut("Could not destroy NUMERIC_DATA msg: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

if (!TipcConnFlush(conn)) {
    TutOut("Could not flush connection to server: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

if (!TipcConnMainLoop(conn, 15.0)) {
    TutOut("Could not run main loop to server: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

TutOut("Destroying connection to server process.\n");
if (!TipcConnDestroy(conn)) {
    TutOut("Could not destroy connection to server: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

return T_EXIT_SUCCESS;
} /* main */

```

Adding Multiple Threads for a Client

The client program listed above is an excellent candidate for use with a server that processes multiple client messages concurrently (such as using multiple threads) because:

- the client can submit multiple messages to the server at once
- the client is not dependent upon the ordering of the responses to its messages
- the server has sufficient resources to handle concurrent client requests

The original example multithreaded server program processes messages from different client connections concurrently, but messages from an individual client are still handled one at a time. By introducing multiple threads for each client connection, overall server throughput can be considerably increased. The built-in synchronization capabilities of the connection make the required modifications straightforward. The relevant portions of the second server program, `connmts2.c`, are shown below.

For the purposes of this example, each client thread creates a fixed number of helper threads.

```
#define HELPER_COUNT 2
```

This function is executed by each of the helper threads:

```
/* ===== */
/*..helper_thread -- thread function for helper threads */
T_PTR T_ENTRY helper_thread(arg)
T_PTR arg;
{
    T_PTR *argv = (T_PTR *)arg;
    T_STR id_str = argv[0];
    T_IPC_CONN client_conn = argv[1];

    TutFree(argv);

    if (!TutTsdSetValue(id_key, id_str)) {
        TutOut("Could not set TSD value for thread: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    if (!TipcConnMainLoop(client_conn, T_TIMEOUT_FOREVER)) {
        TutOut("Could not run main loop for thread: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    return NULL;
} /* helper_thread */
```


The `server_thread` function is modified so that it creates the helper threads when the server thread is connected to a client, and waits for them to exit when the client closes the connection:

```
/* ===== */
/*..server_thread -- thread function for server threads */
T_PTR T_ENTRY server_thread(arg)
T_PTR arg;
{
    T_STRING id_str;
    T_IPC_MT mt;
    T_IPC_CONN client_conn;
    T_INT4 i;
    T_INT4 j;
    T_THREAD thread[HELPER_COUNT];
    T_STRING id_strs[HELPER_COUNT];

    sprintf(id_str, "%d'", *(T_INT4 *)arg);
    ...

    for (i = 0; i < 2; i++) {
        ...

        for (j = 0; j < HELPER_COUNT; j++) {
            T_PTR *argv = (T_PTR *)TutMalloc(sizeof(T_PTR) * 2);

            sprintf(id_strs[j], "%d%c", *(T_INT4 *)arg, 'a' + j);
            argv[0] = &id_strs[j];
            argv[1] = client_conn;
            thread[j] = TutThreadCreate((T_THREAD_FUNC)helper_thread,
                                       argv, NULL);
            if (TutThreadEqual(T_INVALID_THREAD, thread[j])) {
                TutOut("Could not create thread %s: error <%s>.\n",
                      id_strs[j], TutErrStrGet());
                TutExit(T_EXIT_FAILURE);
            }
        }
        if (!TipcConnMainLoop(client_conn, T_TIMEOUT_FOREVER)) {
            /* make sure we reached the end of the data */
            if (TutErrNumGet() != T_ERR_EOF) {
                TutOut("%s: Did not reach end of data: error <%s>.\n",
                      id_str, TutErrStrGet());
            }
        }
    }

    for (j = 0; j < HELPER_COUNT; j++) {
        if (!TutThreadWait(thread[j], NULL)) {
            TutOut("Could not wait for thread %s: error <%s>.\n",
                  id_strs[j], TutErrStrGet());
            TutExit(T_EXIT_FAILURE);
        }
    }
}
```

```

        TutOut("%s: Destroying client connection %d.\n", id_str, i +
1);
        if (!TipcConnDestroy(client_conn)) {
            TutOut("%s: Could not destroy client conn: error <%s>.\n",
                id_str, TutErrStrGet());
        }
    }
    return NULL;
} /* server_thread */

```

Compiling, Linking, and Running

To compile, link, and run the example programs, first you must either copy the programs to your own directory or have write permission in one of these directories:

UNIX:

```
$RTHOME/examples/smrtsock/manual
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]
```

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

Step 1 Compile and link the programs

UNIX:

```

$ rtlink -o connmts1.x connmts1.c
$ rtlink -o connmtc.x connmtc.c
$ rtlink -o connmts2.x connmts2.c

```

OpenVMS:

```

$ cc connmts1.c
$ rtlink /exec=connmts1.exe connmts1.obj
$ cc connmtc.c
$ rtlink /exec=connmtc.exe connmtc.obj
$ cc connmts2.c
$ rtlink /exec=connmts2.exe connmts2.obj

```

Windows:

```

$ nmake /f cts1w32m.mak
$ nmake /f ctcw32m.mak
$ nmake /f cts2w32m.mak

```

To run the programs, start the first or second server process first in one terminal emulator window and then the client process in another terminal emulator window.

Step 2 Start the first server program in the first window**UNIX:**

```
$ connmts1.x
```

OpenVMS:

```
$ run connmts1.exe
```

Windows:

```
$ connmts1.exe
```

To start the second server program instead of the first server program, replace `s1` with `s2`.

Step 3 Start the client program in the second window**UNIX:**

```
$ connmtc.x
```

OpenVMS:

```
$ run connmtc.exe
```

Windows:

```
$ connmtc.exe
```

This is an example of the output from the first server process:

```
Creating server connection.
1: Accepted client connection 1.
1: (message-no 1 job-time 1 submit-time 13:10:16) at 13:10:16
1: (message-no 2 job-time 2 submit-time 13:10:16) at 13:10:17
1: (message-no 3 job-time 7 submit-time 13:10:16) at 13:10:19
2: Accepted client connection 1.
2: (message-no 1 job-time 1 submit-time 13:10:22) at 13:10:23
2: (message-no 2 job-time 2 submit-time 13:10:22) at 13:10:24
2: (message-no 3 job-time 7 submit-time 13:10:22) at 13:10:26
1: (message-no 4 job-time 2 submit-time 13:10:16) at 13:10:26
1: (message-no 5 job-time 1 submit-time 13:10:16) at 13:10:29
1: Destroying client connection 1.
1: Accepted client connection 2.
1: (message-no 1 job-time 1 submit-time 13:10:32) at 13:10:32
2: (message-no 4 job-time 2 submit-time 13:10:22) at 13:10:33
1: (message-no 2 job-time 2 submit-time 13:10:32) at 13:10:33
2: (message-no 5 job-time 1 submit-time 13:10:22) at 13:10:35
1: (message-no 3 job-time 7 submit-time 13:10:32) at 13:10:35
2: Destroying client connection 1.
2: Accepted client connection 2.
2: (message-no 1 job-time 1 submit-time 13:10:40) at 13:10:40
2: (message-no 2 job-time 2 submit-time 13:10:40) at 13:10:41
1: (message-no 4 job-time 2 submit-time 13:10:32) at 13:10:42
```

```

2: (message-no 3 job-time 7 submit-time 13:10:40) at 13:10:43
1: (message-no 5 job-time 1 submit-time 13:10:32) at 13:10:44
1: Destroying client connection 2.
2: (message-no 4 job-time 2 submit-time 13:10:40) at 13:10:50
2: (message-no 5 job-time 1 submit-time 13:10:40) at 13:10:52
2: Destroying client connection 2.
Destroying server connection.

```

Here is an example of the output from each of the client processes connected to the first server:

```

Creating connection to server process.
Reply (message-no 1 job-time 1 submit-time 13:10:16
      in-time 13:10:16 out-time 13:10:17) elapsed 1.122 seconds
Reply (message-no 2 job-time 2 submit-time 13:10:16
      in-time 13:10:17 out-time 13:10:19) elapsed 3.105 seconds
Reply (message-no 3 job-time 7 submit-time 13:10:16
      in-time 13:10:19 out-time 13:10:26) elapsed 10.105 seconds
Reply (message-no 4 job-time 2 submit-time 13:10:16
      in-time 13:10:26 out-time 13:10:29) elapsed 12.108 seconds
Reply (message-no 5 job-time 1 submit-time 13:10:16
      in-time 13:10:29 out-time 13:10:30) elapsed 13.129 seconds
Destroying connection to server process.

```

Here is an example of the output from the second server process:

```

Creating server connection.
1': Accepted client connection 1.
1': (message-no 1 job-time 1 submit-time 13:12:21) at 13:12:21
1a: (message-no 2 job-time 2 submit-time 13:12:21) at 13:12:21
1b: (message-no 3 job-time 7 submit-time 13:12:21) at 13:12:21
1': (message-no 4 job-time 2 submit-time 13:12:21) at 13:12:22
2': Accepted client connection 1.
2': (message-no 1 job-time 1 submit-time 13:12:23) at 13:12:23
2a: (message-no 2 job-time 2 submit-time 13:12:23) at 13:12:23
2b: (message-no 3 job-time 7 submit-time 13:12:23) at 13:12:23
1a: (message-no 5 job-time 1 submit-time 13:12:21) at 13:12:23
2': (message-no 4 job-time 2 submit-time 13:12:23) at 13:12:24
2a: (message-no 5 job-time 1 submit-time 13:12:23) at 13:12:25
1': Destroying client connection 1.
1': Accepted client connection 2.
1': (message-no 1 job-time 1 submit-time 13:12:36) at 13:12:36
1a: (message-no 2 job-time 2 submit-time 13:12:36) at 13:12:36
1b: (message-no 3 job-time 7 submit-time 13:12:36) at 13:12:36
1': (message-no 4 job-time 2 submit-time 13:12:36) at 13:12:37
2': Destroying client connection 1.
1a: (message-no 5 job-time 1 submit-time 13:12:36) at 13:12:38
2': Accepted client connection 2.
2': (message-no 1 job-time 1 submit-time 13:12:39) at 13:12:40
2a: (message-no 2 job-time 2 submit-time 13:12:39) at 13:12:40
2b: (message-no 3 job-time 7 submit-time 13:12:39) at 13:12:40
2': (message-no 4 job-time 2 submit-time 13:12:39) at 13:12:41
2a: (message-no 5 job-time 1 submit-time 13:12:39) at 13:12:42
1': Destroying client connection 2.
2': Destroying client connection 2.
Destroying server connection.

```

Here is an example of the output from each of the client processes connected to the second server: (Note the reduction in message response elapsed times.)

```

Creating connection to server process.
Reply (message-no 1 job-time 1 submit-time 13:12:21
      in-time 13:12:21 out-time 13:12:22) elapsed 1.152 seconds
Reply (message-no 2 job-time 2 submit-time 13:12:21
      in-time 13:12:21 out-time 13:12:23) elapsed 2.153 seconds
Reply (message-no 4 job-time 2 submit-time 13:12:21
      in-time 13:12:22 out-time 13:12:24) elapsed 3.155 seconds
Reply (message-no 5 job-time 1 submit-time 13:12:21
      in-time 13:12:23 out-time 13:12:24) elapsed 3.155 seconds
Reply (message-no 3 job-time 7 submit-time 13:12:21
      in-time 13:12:21 out-time 13:12:28) elapsed 7.15 seconds
Destroying connection to server process.

```

Working With Threads and Connections

Connection programs that call the Tipc* API functions from more than one thread must first call TipcInitThreads, even if the program does not use any SmartSockets threads features. See the reference page for TipcInitThreads in the *TIBCO SmartSockets Application Programming Interface* reference for more details. For example:

```

if (!TipcInitThreads()) {
    /* error */
    TutExit(T_EXIT_FAILURE);
}

```

Connection programs are free to mix and match the portable TutThread*, TutMutex*, TutCond*, and TutTsd* functions with the non-portable native threads functions, such as the POSIX pthreads functions or Win32 threads functions. For more information on the SmartSockets portable multithreading functions, see the *TIBCO SmartSockets Utilities* reference.

The primitive synchronization properties within each connection are used individually or in combinations by the TipcConn* API functions to ensure that threads sharing the connection do not interfere with one another or corrupt the connection's internal state.

Mutex	Type	Default	Description
Read	Mutual exclusion lock C type T_MUTEX	An unlocked mutex if TipcInitThreads has been called first and the architecture supports threads, NULL otherwise.	The read mutex property protects operations that access the connection's message queue, read buffer, or GMD high sequence number table, or read data from the connection's socket.

Mutex	Type	Default	Description
Write	Mutual exclusion lock C type T_MUTEX	An unlocked mutex if TipcInitThreads has been called first and the architecture supports threads, NULL otherwise.	The write mutex property protects operations that access the connection's write buffer or write data to the connection's socket.
Process	Read/write mutual exclusion lock C type T_RW_MUTEX	An unlocked read/write mutex if TipcInitThreads has been called first and the architecture supports threads, NULL otherwise.	The process mutex property protects operations that access the connection's process and default callback lists. A read/write mutex is used here so that multiple threads may execute a connection's process and default callbacks concurrently.
GMD	Mutual exclusion lock C type T_MUTEX	An unlocked mutex if TipcInitThreads has been called first and the architecture supports threads, NULL otherwise.	The gmd mutex property is a temporary mutex that protects operations that access the connection's GMD area. A temporary mutex means that this mutex is never held for an indefinite period of time.
Queue	Mutual exclusion lock C type T_MUTEX	An unlocked mutex if TipcInitThreads has been called first and the architecture supports threads, NULL otherwise.	The queue mutex property is a temporary mutex that protects operations that access the connection's queue such as a message insert or a message delete. A temporary mutex means that this mutex is never held for an indefinite period of time.

Advanced Uses of Connections

The previous section described the most common ways of working with connections. This section shows some advanced ways of using connections.

Mixing Connections and Xt Intrinsics (Motif)

Connections can be used in a Motif (or any other Xt-based widget set) program through the use of the Xt Intrinsics functions `XtAppAddInput` and `XtRemoveInput`. Both Motif and connections have their own main loop functions: `XtAppMainLoop` for Motif and `TipcConnMainLoop` for connections. The easiest way to mix these two main loops is to use `XtAppAddInput` to add the connection as a source of input into Xt's event-handling mechanism and then use `XtAppMainLoop`. One of the parameters to `XtAppAddInput` is the function to be called when data is available. When data is available for reading on the connection, `XtAppMainLoop` calls your user-defined function, which can call `TipcConnMainLoop`. If `TipcConnMainLoop` fails, then `XtRemoveInput` should be called to remove the connection as a source of input. Refer to your operating system manuals for full information on the functions `XtAppAddInput`, `XtAppMainLoop`, and `XtRemoveInput`.

The function `TipcConnGetXtSource` can be used to get an `XtAppAddInput`-compatible source from a connection. On UNIX, this source is the socket file descriptor of the connection (that is, `TipcConnGetXtSource` gets the same thing as `TipcConnGetSocket`). On OpenVMS, this source is an event flag in cluster zero, which is needed for the OpenVMS implementation of `XtAppAddInput`.

This code fragment shows how to use `XtAppAddInput` to register a function to be called when data is available for reading on a connection:

```
/* ===== */
/*..xt_conn_func -- data available on a connection */
void xt_conn_func(client_data, source, id)
XtPointer client_data; /* really (T_IPC_CONN) */
int *source;
XtInputId *id;
{
    T_IPC_CONN conn = (T_IPC_CONN)client_data;

    /* Process all messages that can be read immediately. */
    if (!TipcConnMainLoop(conn, 0.0)) {
        /* an error occurred, so remove the source of input */
        XtRemoveInput(*id);
        /* additional error handling goes here */
    }
}
```

```

} /* xt_conn_func */
/*...code from calling function is below */

T_INT4 xt_source;

if (!TipcConnGetXtSource(conn, &xt_source)) {
    /* error */
}

```



Use `TipcConnMainLoop`, or `TipcConnMsgNext` in a loop here instead of just calling `TipcConnMsgNext` once. Otherwise, messages might be read in but left in the connection message queue where `XtAppMainLoop` cannot see them. Timeout values other than 0.0 should also be used with caution here, as a non-zero timeout can cause the process to wait for data and decrease the responsiveness of the Motif/Xt user interface.

```

XtAppAddInput(app_context,
              xt_source,
#ifdef T_OS_VMS /* needed by OpenVMS XtAppAddInput */
              NULL,
#else
              (XtPointer)XtInputReadMask,
#endif
              xt_conn_func,
              (XtPointer)conn);

```

The previous example uses `TipcConnMainLoop(conn, 0.0)` to read and process all messages that are immediately available on the connection. If your program cannot immediately process all messages for some reason, such as it can only process one message and then must allow the Motif/Xt user interface to be updated, there are many Motif/Xt features available:

- work procedures can be used by calling `XtAppAddWorkProc` to register a function that is called when the user interface event loop is idle
- timeout procedures can be used by calling `XtAppAddTimeOut` to register a function that will be called after a certain amount of time has elapsed
- event manipulation functions like `XtAppPending` and `XtAppProcessEvent` can be used to process additional X events

These Motif/Xt features can be used in a wide variety of combinations, and the best combination depends on the specific needs of the program you are developing. Refer to your operating system manuals for full information on the functions `XtAppAddWorkProc`, `XtAppAddTimeOut`, `XtAppPending`, and `XtAppProcessEvent`.

As described above, using connections with `XtAppAddInput` requires special care for the buffering of incoming messages. Extra effort is also needed sometimes to ensure proper buffering of outgoing messages. `TipcConnMsgSend` sends a message but may not flush the message. Likewise many `TipcMon*` and `TipcSrvSubject*` functions send messages to `RTserver` but do not explicitly flush these messages. This can cause problems because `XtAppMainLoop` does not know to flush these messages, and often no incoming messages arrive until these buffered outgoing messages are flushed. The easiest solutions to this problem are to set the connection auto flush size property to 0 or to call `TipcConnFlush` after sending messages but before returning from your Motif/Xt callback functions.

In addition to checking when data is available to be read from a connection, `XtAppAddInput` can also be used on a server connection to check when a client has connected and can be immediately accepted with `TipcConnAccept`.

The source code for a complete example of mixing SmartSockets and Motif is located in these files:

UNIX:

`$RTHOME/examples/smrtsock/motifipc.c`

OpenVMS:

`RTHOME:[EXAMPLES.SMRTSOCK]MOTIFIPC.C`

Windows:

`%RTHOME%\examples\smrtsock\motifipc.c`

Mixing Connections and the Select Function

The `select` function, which is often associated with sockets but can be used with any file descriptor on most UNIX systems, provides a way to check many file descriptors without polling each one individually. The function `TutSocketCheck` provides a simplified interface to the `select` function that works well if you only have one file descriptor. For example, the function `TipcConnCheck` uses `TutSocketCheck` because a connection has exactly one socket. Many UNIX systems also have a `poll` function that is similar to `select`, and the techniques described in this section can be used for `poll` as well as `select`.

If you need to check multiple connections, or a mixture of sockets and connections, then you must call `select` directly. Refer to your operating system manuals for full information on the function `select`. The function `TipcConnGetSocket` can be used to get the connection's socket file descriptor, which can then be used with `select`.

This code fragment shows how to use `select` to wait for up to 10 seconds for data to be available for reading on a connection or a socket:

```
fd_set read_set;
T_INT4 conn_fd;
T_INT4 largest_fd;
struct timeval time_out;
int status;

/* get the socket file descriptor from the connection */
if (!TipcConnGetSocket(conn, &conn_fd)) {
    /* error */
}

/* set the appropriate bits in the read set */
FD_ZERO(&read_set);
FD_SET(conn_fd, &read_set);
FD_SET(socket_fd, &read_set);

largest_fd = (conn_fd > socket_fd) ? conn_fd : socket_fd;
time_out.tv_sec = 10; /* 10 seconds */
time_out.tv_usec = 0; /* 0 microseconds */

status = select(largest_fd + 1,
                &read_set,
                NULL,
                NULL,
                &time_out); /* use NULL to wait indefinitely */
TutOut("%d file descriptors are ready for reading.\n", status);

if (FD_ISSET(conn_fd, &read_set)) {
    TutOut("Connection is ready for reading.\n");
    if (!TipcConnMainLoop(conn, 0.0)) {
        /* error */
    }
}
```



Use `TipcConnMainLoop`, or `TipcConnMsgNext` in a loop, here instead of just calling `TipcConnMsgNext` once. Otherwise, messages might be read in but left in the connection message queue where `select` cannot see them. Timeout values other than 0.0 should also be used with caution here, as a non-zero timeout can cause the process to wait for data and thus decrease the response time to other processing needs.

```
if (FD_ISSET(socket_fd, &read_set)) {
    TutOut("Socket is ready for reading.\n");
    /* call recv or read */
}
```

In addition to checking when data is available to be read from a connection, `select` can also be used on a server connection to check when a client has connected and can be immediately accepted with `TipcConnAccept`.

Mixing Connections and the Windows Message Loop

At the heart of all Windows applications is a message processing loop. Within this loop, the application checks for the arrival of a Windows message. When one arrives it is processed. The typical call used to check for incoming messages is `GetMessage`. This is a blocking Windows call, and if no message is waiting to be processed, the application will block until a message arrives. To provide timely response to user-generated events, your application must service pending event messages in a timely manner. Interfacing SmartSockets with Windows requires careful coding to ensure that the user interface is serviced in a timely manner while still providing the real-time response required for your application.

Many of the SmartSockets function calls block until a specific condition is met, such as a SmartSockets message arriving. You should be careful not to use these blocking calls in the Windows event-driven environment. Using them can result in your application appearing to be non-responsive to user input, or even, in the case of Win16, locking up the entire system. All of the blocking SmartSockets calls have corresponding non-blocking calls that can be used in an event driven environment.

`WSAAsyncSelect` is a function provided by `winsock.dll`, the Windows Socket Library. It provides a means for receiving notification when a SmartSockets message arrives on a connection. The service is activated by passing the connection's socket handle along with an indication of what events are to be notified; in this case you want to know when data is ready to be read. The SmartSockets function `TipcConnGetSocket` returns the connection's socket handle. A Windows message number and `HWND` window handle to notify are also passed to the function. When data is ready to be read from the socket, a message is sent to your window.

The message number passed to the function should be a user-defined message, based on `WM_USER`. The socket notification remains in force until explicitly cancelled. This code fragment shows how to use `WSAAsyncSelect` to wait for data to be available for reading on a connection:

```
#define MY_SOCKET_MSG (WM_USER + 0)

T_IPC_CONN conn;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                  LPSTR lpszArgs, int nWinMode)
{
    HWND hwnd
    SOCKET sock;

    /* Define a window class */
    ...

    /* Register the window class */
    ...
```

```

/* Create the window */
hwnd = CreateWindow(...

/* Create client connection */
conn = TipcConnCreateClient(...)

/* get the connection's socket */
if (!TipcConnGetSocket(conn, &sock)) {
/* error */
}

/* set up for read notification */
if (WSAAsyncSelect(sock, hwnd,
MY_SOCKET_MSG, FD_READ) == SOCKET_ERROR) {
/* error */
}

/* main message loop */
...
}

LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
WPARAM wParam, LPARAM lParam)
{
switch (message) {
case ...
case MY_SOCKET_MSG:
if (!TipcConnMainLoop(conn, 0.0)) {
/* error */
}
break;
default: ...
}
}
}

```

In addition to checking when data is available to be read from a connection, `WSAAsyncSelect` can also be used on a server connection to check when a client has connected and can be immediately accepted with `TipcConnAccept`.

The source code for a complete example of mixing SmartSockets and the Windows Microsoft Foundation Classes (MFC) is located in these directories:

UNIX:

```
$RTHOME/examples/smrtsock/windows/rtwcon
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.WINDOWS.RTWCON]
```

Windows:

```
%RTHOME%\examples\smrtsock\windows\rtwcon
```

Remote Procedure Calls

Messages can be sent between connections to perform a remote procedure call. A remote procedure call (RPC) is a means for a process to execute a function in another process and wait for the result of the function call. Normally when a process sends a message through a connection with `TipcConnMsgSend`, the sending process continues and does not wait for the receiving process to receive the message and act on it. This normal mode of operation can be thought of as a non-blocking RPC that does not return a result.

The function `TipcConnMsgSendRpc` performs a blocking RPC that does return a result. One message is sent as the RPC call from the caller end of the connection, and another message is sent back as the RPC result to the caller. The callee end of the connection must be prepared to receive the call message and send back the result message. `TipcConnMsgSendRpc` uses a simple relationship between the call and result messages: the message type number of the result message must be one greater than the message type number of the call message.

See the reference information for `TipcConnMsgSendRpc` in the *TIBCO SmartSockets Application Programming Interface* for a code example of how to perform RPCs with connections.

Time Resolution

Many connection properties, such as read timeout, and API functions, such as `TipcConnMsgNext`, have a time-related value associated with them. Most SmartSockets time-related values are stored in a `T_REAL8` eight-byte real number. A `T_REAL8` has a limited precision of approximately 15 significant digits. Many of the time-related values are further constrained by the resolution provided by the `select` function and the resolution of the operating system internal clock. The `select` function uses a `timeval` structure similar to the SmartSockets `T_TIME_STRUCT` structure; both have two four-byte integer fields for fixed second and microsecond resolution. Most operating systems do not provide microsecond resolution, however. See *Mixing Connections and the Select Function* on page 143 for more information on using connections with `select`. Time-related values can always use fractional seconds but the resolution of the fractional part varies depending on your configuration.

File Descriptor Upper Limit

All operating systems supported by SmartSockets have an upper limit on the number of file descriptors and socket descriptors a process can use simultaneously. On most UNIX-based operating systems the `setrlimit` function is used to raise the per-process descriptor limit. On OpenVMS the `FILLM` quota controls the per-process descriptor limit. Refer to your operating system manuals for more information on configuring descriptor limits to larger values (sometimes the operating system itself must be reconfigured).

The size of the `fd_set` socket data structure controls how many socket descriptors can be used with the `select` function, regardless of the setting of the per-process descriptor limit. The `fd_set` data structure is used by `TutSleep`, `TutSocketCheck`, `RTlm`, and `RTserver`. The size of `fd_set` is by default defined in an operating system C language header file, is compiled into SmartSockets, and cannot be changed at run-time.

On most operating systems supported by SmartSockets, `fd_set` is large enough to accommodate at least 1024 simultaneous descriptors. This allows a process like `RTserver` the potential to manage approximately 1015 `RTclient` processes at once! On some platforms SmartSockets has direct support for the `poll` system call. On these platforms the number of descriptors that can be simultaneously managed by `rtserver` is greatly increased. To support large numbers of client connections in `RTserver`, the option `Max_Client_Conns` must be increased above its default value.

Handling Network Failures

SmartSockets has been designed to handle many different kinds of network failures, and this robust behavior provides a certain level of fault tolerance. The core function of SmartSockets fault tolerance is in connections.

This section describes the features of connections that implement fault tolerance. For a discussion of the features specific to RTserver and RTclient that add more fault tolerance, such as hot switchover from primary RTclients to backup RTclients, see *Handling Network Failures In Publish Subscribe* on page 307, and *Running an RTclient With a Hot Backup* on page 429.

In addition to detecting network failures, connections can completely recover from these failures by using guaranteed message delivery, covered in Chapter 4, *Guaranteed Message Delivery*.

What is Fault Tolerance?

Fault tolerance is a term used to describe computer systems that continue to function even when some of the hardware and software fail. Examples of failure conditions include:

- processes or computers running out of memory
- processes hanging or going into infinite loops
- computers crashing or hanging
- breaks in network cables
- misconfigured computers
- overloaded computers causing processes to run slowly

Fault tolerance can be implemented in hardware by mirrored filesystems on multiple disks, redundant networks, redundant CPUs, redundant memory, and so on. Hardware-based fault tolerant systems are more expensive than non-fault tolerant systems due to the extra components. Fault tolerance can also be implemented in software by products such as SmartSockets. Surprisingly, enabling the fault tolerant features of connections has little effect on message throughput.

The general mechanisms that SmartSockets uses for fault tolerance are:

- avoid operations that can block indefinitely, or put an upper limit on the amount of time these operations can block
- periodically check for potential failure conditions

Potential Network Failures

Connections use sockets as the communication link between two processes, and thus can use the features of sockets and each network protocol for detecting network failures. There are three areas of connections where problems can occur:

- creating a connection
- sending data on a connection
- receiving data on a connection

In each area SmartSockets builds on top of the features of sockets and network protocols to provide faster detection of network problems. Each IPC protocol (local, and TCP/IP) handles failures differently, which complicates matters. For the local protocol, there are no possible network failures, because this protocol does not use a network, although processes that use the local IPC protocol can still fail.

For creating a connection, the server connection does not need any special handling because creation of the server connection with the function `TipcConnCreateServer` completes immediately. The process with the server connection can use `TipcConnCheck` to check if a client has connected before calling `TipcConnAccept` to accept the client connection. The creation of the client connection with the function `TipcConnCreateClient` may not complete immediately if the node of the server connection is somehow unavailable, such as it has crashed, is turned off, or the network is ruptured. For TCP/IP client connections, the option `Socket_Connect_Timeout` can be used to set a limit on how long (in seconds) to wait for availability. The default value for `Socket_Connect_Timeout` is 5.0. If `Socket_Connect_Timeout` is set to 0.0, then the client connection creation timeout is disabled, and TCP/IP clients block for an operating system-dependent amount of time if the server node is not available (typically 75 seconds for TCP/IP, for example).

When sending data on a connection, if the data cannot be sent, either the receiving process is not keeping up or a network failure has occurred. The TCP/IP protocols by default do not send any packets during periods of inactivity and do not forcefully break a link for many types of network problems (for example, a broken network cable). TCP/IP does have the concept of an optional keepalive that can be enabled. This network-level TCP/IP keepalive is different from an application-level connection keep alive, but serves the same purpose. From this point, the term keepalive (one word) is used to refer to a TCP/IP health check, and the term keep alive (two words) is used to refer to an application-level health check. The default TCP/IP keepalive timeout is very large on most systems (typically two hours), cannot be changed by non-privileged users, and can only be changed for all TCP/IP links, not just one. This makes the TCP/IP keepalive

unusable for most applications. It is available to SmartSockets programs though, through the socket option (not to be confused with a SmartSockets option) `SO_KEEPAVIVE`. Refer to your operating system manuals for full information on this socket option.

This code fragment enables TCP/IP keepalives on a connection's socket:

```
T_INT4 conn_socket;
int one = 1;
if (!TipcConnGetSocket(conn, &conn_socket)) {
    /* error */
}
if (setsockopt(conn_socket, SOL_SOCKET, SO_KEEPAVIVE,
               (char *)&one, sizeof(one)) != 0) {
    /* error */
}
```

For receiving data on a connection, if data cannot be received, then either the sending process has not sent anything, or a network failure has occurred. The above-mentioned features of TCP/IP also apply for receiving data: TCP/IP does not send packets during periods of inactivity (by default). If no data is received within a certain period of time, a connection can initiate a connection keep alive (not to be confused with a TCP/IP keepalive) to check the health of the connection. Keep alives are discussed in detail in the next section.

Keep Alives

A connection keep alive is a very simple way to check the health of a connection, including the network and the process at the other end of a connection. The function `TipcConnKeepAlive` is used to perform a keep alive. Connection keep alives are remote procedure calls that send a `KEEP_ALIVE_CALL` message through a connection and then wait for a `KEEP_ALIVE_RESULT` message back from the other process. If the other process is alive, it receives the `KEEP_ALIVE_CALL` message and sends back a `KEEP_ALIVE_RESULT` message. If the keep alive originator does not receive a response within a certain period of time, it assumes there has been a network failure and destroys the connection or takes other actions.

For most uses, you can simply set the block mode, read timeout, write timeout, and keep alive timeout properties of a connection to automatically control checking for network failures (see Connection Composition on page 71 for details). The function `TipcConnCheck` automatically calls `TipcConnKeepAlive` if the amount of time that has elapsed since data was last read from the connection is greater than the read timeout property of the connection. A connection by default processes a `KEEP_ALIVE_CALL` message with the process callback function `TipcCbConnProcessKeepAliveCall`. This function handles sending back

a `KEEP_ALIVE_RESULT` message to the process that originated the keep alive. While timeout checking is normally done automatically and transparently by `TipcConnCheck`, you can call `TipcConnKeepAlive` directly to explicitly check the health of a connection.

You should not try to explicitly send or receive `KEEP_ALIVE_CALL` and `KEEP_ALIVE_RESULT` messages, but instead always use `TipcConnCheck`, `TipcConnKeepAlive`, and `TipcCbConnProcessKeepAlive` to handle the details of keep alives. Because keep alives are checking the health of both the network and the other process, a process must be careful to read and process messages at a regular interval; otherwise the keep alives fail.

Blocking and Non-Blocking Read/Write Operations

As described in Block Mode on page 74, for read timeouts, write timeouts, and automatic keep alives to be enabled, the connection block mode must be set to `FALSE` to enable non-blocking read and write operations. If the connection block mode is `TRUE`, then read and write operations can block indefinitely, and many network failures cannot be detected.

Connection read and write operations are handled differently. If no data can be read within a certain period of time, some kind of failure may have occurred, or there may simply be no data to read. Thus if a read timeout occurs, a keep alive is initiated to check if the process at the other end of the connection is still alive.

If no data can be written within a certain period of time, however, this indicates a problem, as the connection's socket is full. There is no point in initiating a keep alive when a write timeout occurs because the keep alive RPC call will most likely not be able to be written to the already-plugged socket.

This chapter introduces RTserver and RTclient, which allow many processes to easily communicate with each other using a publish-subscribe communication model. This model is different than the peer-to-peer model described in Chapter 2, Connections.

Topics

- *Publish-Subscribe Overview, page 154*
- *RTserver and RTclient Composition, page 156*
- *TIBCO SmartSockets Multicast, page 170*
- *Essential API Functions, page 172*
- *Working With RTclient, page 175*
- *Message File Logging, page 210*
- *Load Balancing, page 215*
- *Using Threads with the RTclient API, page 226*
- *Advanced RTclient Usage, page 227*
- *Using a Dispatcher, page 249*
- *Message Compression, page 271*
- *Security, page 275*
- *Starting and Stopping RTserver, page 284*
- *Working with RTserver, page 290*
- *Dynamic Message Routing, page 296*
- *Network Considerations, page 305*

Publish-Subscribe Overview

An RTserver process routes messages between RTclient processes. A key feature of SmartSockets is the ability to distribute RTserver and RTclient processes over a network. Different processes can be run on different computers, taking advantage of all the computing power a network has to offer. RTservers and RTclients can be dynamically started and stopped while the system is running.

The feature set of SmartSockets publish-subscribe architecture is layered on top of connections and messages, but adds greater function and ease of use. Some of these functions are listed below.

- RTserver and RTclient have simplified setup and control through options, which require no programming.
- RTserver can partition a group of RTclients into a project.
- RTclient and RTserver use logical addresses called subjects for the sender property and destination property of messages, which enable powerful yet simple publish-subscribe services.
- Subjects are arranged in a hierarchical namespace with wildcard capabilities, which makes it easier to build large projects.
- A group of RTservers can distribute the load of publish-subscribe message routing with dynamic message routing that offers greater scalability and flexibility.
- Routing between RTservers can be further optimized to use the lowest cost route by assigning a cost to each path between RTservers.
- Additional callbacks are available including subject, server create, and server destroy.
- An RTclient can automatically start an RTserver, automatically restart an RTserver, and even continue running when an RTserver is temporarily unavailable.
- RTserver and RTclient have advanced guaranteed message delivery (GMD) capabilities, such as automatic recovery from most network failures.
- Much information about RTservers and RTclients can be monitored, which allows you to easily debug, examine, and control your projects.
- Load balancing can be used to publish messages to one subscriber instead of all interested subscribers.
- The RTclient API can be used safely in multithreaded programs.

This chapter describes RTserver and RTclient composition, how to work with the RTclient Application Programming Interface (API), and how to work with RTserver. Monitoring is mentioned briefly here. See Chapter 5, Project Monitoring, for a full discussion on monitoring.

Most RTclients have a single, global connection to one RTserver. Generally, this is the only connection required, and this connection forms the basis for most RTclient-RTserver interactions. The focus of this chapter is on understanding this connection and the interactions involved. You should be thoroughly familiar with this connection and the relevant APIs before delving into more complex connections. Connecting to an RTgms process instead of an RTserver process for the single, global connection requires an understanding of multicasting and a special set of PGM options. For more information, see TIBCO SmartSockets Multicast, page 170. It is also possible to connect to multiple RTservers, using a special multiple connection instead of the global connection. However, this type of connection requires a completely different set of APIs, and each individual connection must be created and configured discretely. For more information, see Connecting to Multiple RTservers, page 248.

RTserver and RTclient Composition

Before you use RTserver and the RTclient API, it helps to have an understanding of the concepts involved:

RTserver	is a process that extends the features of connections to provide transparent publish-subscribe message routing among many processes.
RTclient	is any program (user-defined or SmartSockets client) that connects to RTserver and accesses its services (under this definition RTmon can be considered an RTclient).
Project	is a group of RTservers which exchange messages only with other RTclients in the same project.
Subject	is a logical address for a message. RTclient subscribes to subjects, registering interest in those subjects. An RTclient also publishes messages to subjects, meaning the RTclient sends messages to subjects.
Data Frame	is a group of messages with the same timestamp (not shown in the figure).
Monitoring	allows you to examine detailed information about your project in real time (not shown in the figure).

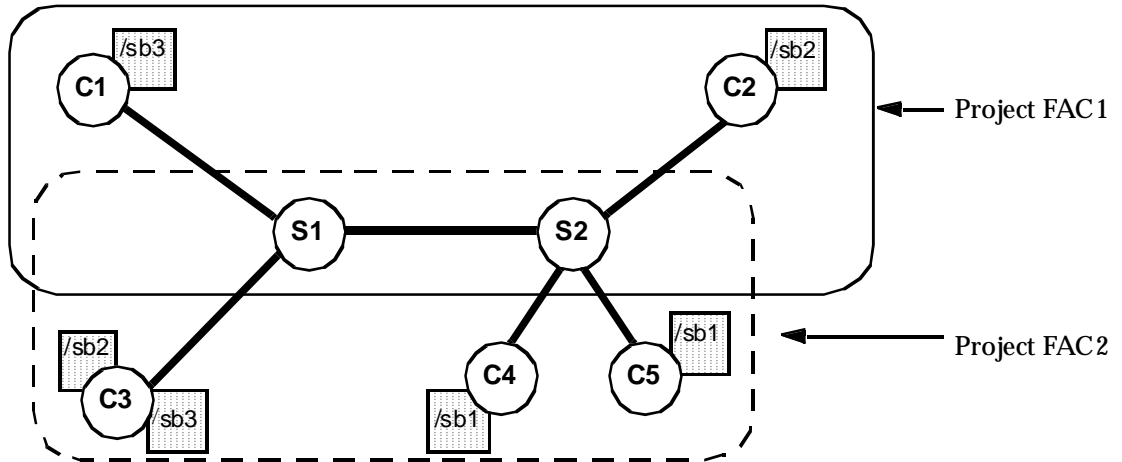
Projects

Many RTclients can simultaneously use the same computer, and RTclients on different computers can also send messages to each other. Two RTclients on a network may or may not want to receive each other's messages. Projects provide a way to distinguish between groups of SmartSockets processes. A group of RTservers exchange messages only with other RTclients in the same project. For SmartSockets, a project is typically used to acquire, analyze, archive, and display data. A project is designated by a name, which must be an identifier. For example, one common project name is the name of your company's product or application.

A project is a self-contained unit or partition that prevents unwanted messages from being sent to processes, in that RTclients in different projects cannot send messages to each other. Typically, an RTclient belongs to only one project, while an RTserver can provide publish-subscribe routing services for one or more projects. It is possible for an RTclient to connect to more than one project in the same RTserver or to multiple projects across RTservers. See [Connecting to Multiple RTservers](#) on page 248 for more information.

For example, if RTclients are separately monitoring two factories, and the processes are running on the same network, two projects can be used to ensure that messages are not sent between the two separate factory projects. As shown in Figure 11, these projects could be named FAC1 and FAC2. The option `Project` is used to specify the project to which an RTclient belongs. The default value for `Project` is `rtworks`. You should always set this option to prevent becoming part of the default `rtworks` project, which may cause unwanted messages to be received.

Figure 11 RTserver and RTclient Architecture



Note





Project FAC1 has processes C1, S1, S2, and C2.

Project FAC2 has processes C3, S1, C4, S2, and C5.

Both RTserver S1 and S2 are used by both projects.

RTclient C1 cannot send messages to C3, C4, or C5 because they are not in the same project.

A message published (sent) to the /sb1 subject in project FAC2 is received by both RTclient C4 and C5.

-  = a connection
-  = RTserver process
-  = RTclient process
-  = a subject being subscribed to by RTclient

Subjects

Just as projects restrict the boundaries of where messages are sent, subjects can also further partition the flow of messages in a project. A subject is a logical message address that can be thought of as providing a virtual connection between RTclients. Subjects allow an RTclient to send a message to many processes with a single publish operation. Subjects are designated by a name, which can be any character string with a few restrictions (see Hierarchical Subject Namespace, page 159 for details).

The following sections discuss the properties of subjects and the operations that can be performed with subjects. The programming aspects of these operations are described in detail in Using Subjects, page 203.

Subscribing to a Subject

As described in Chapter 1, Messages, a message has both a sender property and a destination property. When the TipcConn* functions are used to send messages through connections, the sender and destination properties are by default not used. There are no predefined values for these properties when working with connections.

For RTserver and RTclient, though, subjects are used for these properties. When an RTclient is subscribing to a subject, it receives any published messages (sent to RTserver) whose destination property is set to that subject. For example, in a satellite project, you might partition messages by functional area — electrical power, thermal, pointing control, and so on. These areas are declared as subjects such as /elec_pwr, /thermal, /pt_ctrl. All messages pertaining to electrical power are constructed with the /elec_pwr subject as their destination property. Any RTclient interested in receiving messages destined for /elec_pwr subscribes to the /elec_pwr subject. This is also known as the publish-subscribe paradigm in that RTclients publish messages to specific subjects, and RTclients subscribe to subjects in which they are interested.

The core capability of SmartSockets publish-subscribe is conceptually similar to how electronic mail (email) mailing lists operate. A person sends an email message addressed to the list, and the message is delivered to all persons subscribed to the list. SmartSockets publish-subscribe is much more powerful, however, as it offers much higher performance, more dynamic capabilities, monitoring, and so on.

When two processes create any kind of peer-to-peer connection to each other, including through T_IPC_CONN connections, they need specific physical network addresses, such as for TCP/IP a node number and port number, to begin communicating. If a process wants to send a message to many other processes, it needs first to know the physical network addresses of the other processes, and then to create connections to all of those processes. This kind of architecture does

not scale well, as configuration is complicated and tedious. The RTserver and RTclient architecture's use of subjects for message addresses allows an RTclient to simply send the message with a subject as the destination property, and RTserver takes care of routing the message to all RTclients that are subscribed to that subject. RTclient can start or stop subscribing to a subject at any time, which allows RTclient to control the quantity of incoming data as desired.

Hierarchical Subject Namespace

To provide greater flexibility and scalability for large projects, SmartSockets subject names are arranged in a hierarchical namespace much like UNIX file names or World Wide Web Universal Resource Locators (URLs). This hierarchical namespace allows for large numbers of subject names to be created with similar but not conflicting names, such as `/system/primary/elec_pwr` and `/system/backup/elec_pwr`, and also for many powerful operations, such as publish-subscribe with wildcards, to be performed. Small SmartSockets projects can be built without requiring large amounts of complexity, and large projects can also be more easily built with these hierarchical subject names.

A hierarchical subject name consists of components laid out left-to-right separated by forward slashes (/). Each component can contain any other non-slash characters except asterisks (*) and the ellipsis (...), both of which are used for wildcards in publish-subscribe. Examples of hierarchical subject names include `/system`, `/system/primary/eps`, `/system/backup/eps` and `/nodes/workstation.tibco.com/support`.

Generally, a subject name is unlimited in length. However, each individual component of a hierarchical subject name is limited to 63 characters in length, due to restrictions imposed when certain commands, such as `subscribe`, `unsubscribe`, or `setopt`, are processed.

An absolute subject name starts with a forward slash (/). SmartSockets allows for greater flexibility and easier configuration by allowing any combination of absolute and non-absolute subject names used in a project. All non-absolute subject names automatically have the value of the option `Default_Subject_Prefix` prefixed to them so as to create a fully qualified name for the hierarchical subject namespace. This allows projects written using non-absolute subject names to be easily moved from one area of the hierarchical subject namespace to another, such

as from `/company/new-york` to `/company/san-francisco`, by simply changing the value of the option `Default_Subject_Prefix`. Both `RTclient` and `RTserver` have the option `Default_Subject_Prefix`. If the option is not set in the `RTclient` (that is, has the value `unknown`), the `RTclient` inherits the `Default_Subject_Prefix` from the first `RTserver` it connects to.



From this point on, any non-absolute subjects used in examples should be treated as if they have `Default_Subject_Prefix` prefixed to them when they are used by `RTclient`.

Subject Wildcards For Publish-Subscribe

Using wildcards (`*` or `...`) in subjects is much like using wildcards for file names in an operating system command line. The asterisk wildcard operates much as it does on Windows, UNIX, OpenVMS, or MVS ISPF environments. It can be used for an entire subject name component or as part of a more complicated wildcard containing other characters, such as `foo*bar`. A wildcard component using an asterisk (`*`) never matches more than one component, such as `foo*bar` does not match `foo/bar`.

The ellipsis wildcard operates much as it does on OpenVMS, where it matches any number of levels, including zero levels, of components. It must be used as an entire component (that is, `auto...` is not a wildcard). Multiple wildcards can be used in a subject name, such as `/a*b*/.../d`. The following table shows several wildcarded subjects with examples of matches and mismatches.

Table 5 Wildcard Subject Examples

Wildcard Subject	Matches	Does Not Match
/stocks/auto/...	/stocks/auto	/stocks
	/stocks/auto/*	
	/stocks/auto/ford	
	/stocks/*/ford	
/sports/*/sanjose	/sports/nhl/sanjose	/sports/nhl/*/coach
	/sports/abl/sanjose	
	/.../sanjose	
/stocks/a*/...	/stocks/auto	/stocks/computer
	/stocks/*/...	
/personnel/.../fred	/personnel/eng/fred	/personnel
	/personnel/sales/mgmt/fred	
	/...	

Table 5 Wildcard Subject Examples

Wildcard Subject	Matches	Does Not Match
/sports/*/...	/sports/baseball/sfgiants /sports/football/sfniners /sports/...	/sports

When a message is published, if multiple subscribed-to subjects match, only one copy of the message is delivered to each subscribing RTclient. An example use of wildcard subscribes is to subscribe to a wildcard subject such as /stocks/auto/... so as to receive all messages published to the non-wildcard subjects that match. An example use of wildcard publishes is to publish to a wildcard subject such as /stocks/auto/... so as to send messages to all subscribers of the non-wildcard subjects that match. An easy way to publish a message to all RTclients in the entire project is to publish the message to the subject "/...". RTserver caches the matching values for wildcard subjects so that potentially time-consuming wildcard matching is not needed for each message it routes.

There is some overlap between projects and hierarchical subjects in that both can be used to partition publish-subscribe, but projects should be used when it is certain that no intercommunication should be allowed. Projects are also useful for having the same subject names simultaneously in both a testbed application and a production application.

For more details on using hierarchical subjects in large-scale projects, see *Dynamic Message Routing*, page 296.

Monitoring a Subject

RTclient can also monitor many things about a subject, such as:

- the names of all subjects in a project
- the RTclients that are subscribed to a subject
- the subjects that an RTclient is subscribing to
- message traffic statistics

This is useful for monitoring process activity. RTclient can start or stop monitoring a subject at any time. Monitoring subjects is discussed in Chapter 5, *Project Monitoring*, and is shown in detail in the example code in *Running an RTclient With a Hot Backup*, page 429.

Unique Subject

Each RTclient has a unique subject that is always used as the sender property of a message sent to RTserver. When an RTclient first connects to RTserver, it automatically subscribes to its unique subject. RTserver does not allow multiple processes to have the same unique subject. By using the unique subject as the sender property, an RTclient that receives a message can easily determine who sent the message and also what subject to use to reply to the message if needed. The option `Unique_Subject` is used to specify the unique subject of an RTclient. The default value for `Unique_Subject` is `_Node_Pid`, where *Node* is the network node name of the computer on which the process is running, and *Pid* is the operating system process identifier of the process.

The `Unique_Subject` option is also used to configure GMD and monitoring. For GMD, `Unique_Subject` is used to construct the pathnames for GMD disk files (see *Configuring GMD*, page 331 for details). For monitoring, `Unique_Subject` is used to identify all RTclients and RTservers.

Standard Subjects

Because subjects are an integral part of publish-subscribe communication, several standard subjects provide a way to consolidate some useful subject-based operations. The standard subjects are operated on as a set. The standard subjects are:

user-defined subjects Subjects listed in the option `Subjects`. These can be considered user-defined standard subjects.

`_Node` The node-specific subject, where *Node* is the network node name of the computer on which the process is running. The `_Node` subject provides a way to publish a message to all RTclients running on a specific node.

`_all` Common subject for all RTclients. The `_all` subject provides a way to publish a message to all RTclients on all nodes.

`_Process` Standard SmartSockets process type subject (`_mon`). User-defined RTclients do not have this process type subject. The process type subject provides a way to publish a message to all standard SmartSockets processes of a certain type.

These operations can be performed in a single step on the standard subjects:

- start subscribing to the standard subjects
- stop subscribing to the standard subjects
- retrieve the standard subjects

User-defined subjects that are not listed in the Subjects option may also be used at any time by using API functions, such as `TipcSrvSubjectSetSubscribe`, or commands, such as `subscribe`, through a control channel.

What is RTserver?

While connections provide a means for two processes to exchange messages, connections by themselves do not scale well to many processes. RTserver fills this void and expands the capabilities of connection-based message passing; RTserver is a publish-subscribe message router that uses connections to make large-scale distributed IPC easier. RTserver also has advanced GMD and monitoring function.

RTserver runs as a background process (on OpenVMS and Windows this is known as a detached process, on MVS it can be an STC or started task) without an interactive command interface. You can start RTserver manually from the operating system prompt, or it can be started automatically when an RTclient first tries to connect to RTserver on most platforms.

To start and use an RTserver, you must have a license for that RTserver. The license needs to be added to the license file, `talarian.lic`, in the standard directory. Or you can brand the RTserver using the `rtbrand` command. The information in the license file takes precedence over any branding that you might do. If you are upgrading an existing license to add more RTservers or to change the type, such as from a single threaded to a multi-threaded RTserver, you must add the new license information to the license file or else re-brand the RTservers. See the *TIBCO SmartSockets Installation Guide* for information on the license file and on branding.

In addition to routing publish-subscribe messages between RTclients, multiple RTservers can route messages to each other. Multiple RTservers can distribute the load of message routing. If a project is partitioned so that most of the messages being published are routed between RTclients on the same node, then the use of multiple RTservers can reduce the consumption of network bandwidth, because processes on the same node can use the non-network local IPC protocol. See Dynamic Message Routing on page 296 for more information on multiple RTservers.

Most project and subject information is kept in RTserver. RTserver maintains a table of some RTclients (including warm RTclients, which are discussed in Warm RTclient in RTserver, page 346), a table of all projects, and a table of some of the subjects in each project. These tables contain information about RTclients subscribing to a subject, the information being monitored by RTclients, and which other RTservers have RTclients subscribing to a subject. Each RTserver knows only

the information necessary for its own publish-subscribe operation. Each RTserver does not know about all RTclients. This greatly increases the scalability of SmartSockets publish-subscribe projects. These scalability features are discussed in more detail in Dynamic Message Routing, page 296.

When RTserver is running but not being used, it uses very little CPU time. The amount of memory used by RTserver varies depending on the number of other (RTserver and RTclient) processes RTserver has connections to and the number of messages that RTserver has buffered to be routed to other RTservers and RTclients. RTserver also deallocates memory for defunct subjects (a subject becomes defunct when no more processes are subscribed to it) so that its memory consumption does not grow over long periods of time. For more information, refer to Starting and Stopping RTserver, page 284.

Publish-Subscribe Message Routing Example

This section provides an example of how a message originating from a single RTclient is published (sent) to RTserver and routed to all RTclients subscribing to a subject. In this example, there are three RTclients: RTclient1, RTclient2, and RTclient3. Each of these RTclients is connected to the same RTserver. RTclient2 and RTclient3 are both subscribed to the /sb1 subject. If RTclient1 publishes a message to the /sb1 subject, this sequence of events occurs:

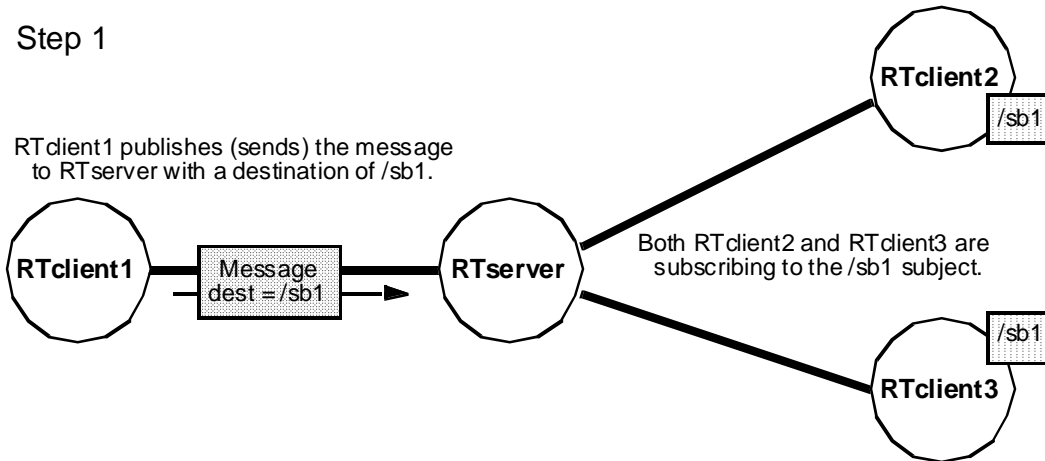
1. RTclient1 constructs a message with /sb1 as the destination.
2. RTclient1 sends the message to RTserver.
3. RTserver receives the message.
4. RTserver looks at the destination (/sb1) of the message.
5. RTserver sends the message to all RTclients currently subscribing to the /sb1 subject.
6. RTclient2 and RTclient3 each receive a copy of the message.

Figure 12 shows this flow of the message through RTserver. Note that if RTclient1 also subscribes to the /sb1 subject prior to publishing the message to the /sb1 subject, it too receives the message from RTserver.

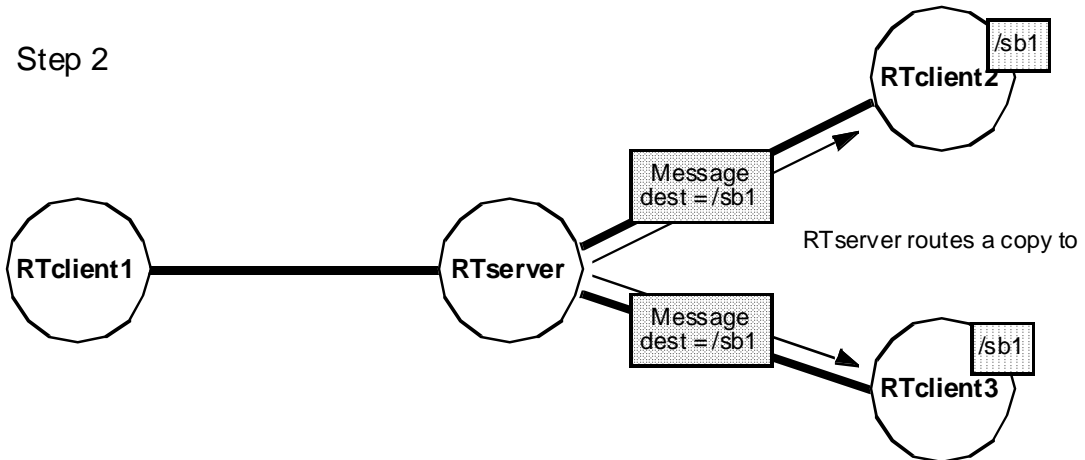
Figure 12 RTserver Publish-Subscribe Message Routing

Step 1

RTclient1 publishes (sends) the message to RTserver with a destination of /sb1.



Step 2



Monitoring RTserver

RTclient can monitor many things about RTserver. Some of these are listed below.

- the names of all RTservers in an RTserver group, a server cloud
- generic information about an RTserver, such as what node it is running on
- buffer statistics for an RTserver, such as message backlog
- option values in an RTserver, such as current configuration
- time information in an RTserver
- message traffic statistics in an RTserver
- inter-RTserver connection topology information
- routing information in an RTserver

This is useful for monitoring process activity. RTclient can start or stop monitoring an RTserver at any time. Monitoring RTserver is discussed in Chapter 5, Project Monitoring.

What is RTclient?

An RTclient is a process that is connected to an RTserver as a client. Usually, each RTclient has one T_IPC_CONN connection to one RTserver, because an RTclient cannot have more than one global connection at a time. In rare instances where an RTclient requires multiple connections to RTservers, that RTclient can use multiple RTserver connections. These are described in more detail in [Connecting to Multiple RTservers](#), page 248.

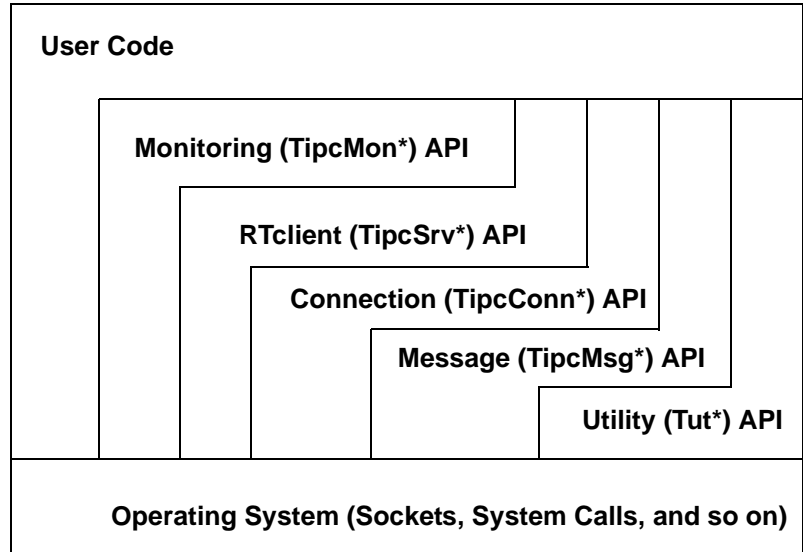
An RTclient can send messages, receive messages, and create callbacks using the global connection just as it would any other connection. The publish-subscribe message routing capabilities of RTserver are transparent to an RTclient, and subjects provide a virtual connection between RTclients.

An RTclient can have complete control over when it creates a connection to RTserver, or it can automatically create the connection when it is first needed. If an RTserver is not already running, an RTclient can start an RTserver. If an unrecoverable error (such as a network failure) occurs, the RTclient can restart the RTserver.

All of the capabilities of the TipcConn* functions are available to an RTclient, including most of the connection callback types (all except encode and decode). An RTclient can partially destroy its connection to RTserver and temporarily continue running as if it were still connected, or an RTclient can fully destroy its connection to RTserver and continue as if it had never been connected at all. An RTclient using GMD can even crash, be restarted, and recover from most network failures.

Because of some extra function in RTclient, you do not use the TipcConn* functions, but instead use a parallel set of TipcSrv* functions. Almost every TipcConn* function has an equivalent TipcSrv* function. Figure 13 shows the relationships among the various IPC functions.

Figure 13 The Layers of the SmartSockets API



See Working With RTclient on page 175 for a full discussion of the TipcSrv* functions.

Monitoring RTclient

RTclient can monitor many things about any RTclient, such as:

- the names of all RTclients in a project
- the subjects that an RTclient is subscribing to
- messages received by an RTclient
- messages sent by an RTclient
- data created by an RTclient, referred to as extension data
- generic information about an RTclient, such as what node it is running on
- buffer statistics for an RTclient, such as message backlog
- option values in an RTclient, such as current configuration
- time information in an RTclient
- message type, callback, and subject information and statistics about an RTclient

This is useful for monitoring process activity. RTclient can start or stop monitoring an RTclient at any time. Monitoring RTclients is discussed in Chapter 5, Project Monitoring.

Peer-To-Peer Or Client-Server

The publish-subscribe architecture of RTserver and RTclient is implemented using SmartSockets peer-to-peer connections and takes advantage of all the features that connections have to offer. Sometimes it can be unclear when to use a connection to send and receive messages, and when to use RTserver to send and receive messages. A general guideline is to use just connections when communicating between exactly two processes (that is, a peer-to-peer situation) and to use RTserver otherwise. These scenarios illustrate this guideline in detail:

- Use connections when only two processes need to communicate; subjects are unnecessary in this case.
- Use RTserver when there are a varying number of processes involved in the task. For example, if a process needs to communicate equally well with 1, 10, or 20 other processes, using RTserver instead of just connections can greatly simplify the development task.
- Use RTserver when sending messages to or receiving messages from a standard SmartSockets process (except for a data source). For example, if an RTclient wants to send a message to all RTclients that are currently running, it uses publish-subscribe through RTserver instead of trying to find and create connections to all RTclient processes.
- Use RTserver on platforms which do not support threads. A parent process (task) can give work to child processes (subtasks) and the RTserver can be used to route the results back to the requestor.
- Use RTservers when protocol bridging is required. If there is not a consistent network protocol that all processes can use, then RTservers can be used judiciously to route messages between heterogeneous network segments.

Keep in mind that a process can use as many connections as it wants and can mix TipcConn*, TipcSrv*, and TipcMon* functions as needed.

Ease of Use

RTclient and RTserver can be controlled using both options and API functions. Throughout this chapter, both approaches are documented as appropriate. All RTclient and RTserver options are documented in Chapter 8, Options Reference. The API functions are documented in the *TIBCO SmartSockets Application Programming Interface* reference.

TIBCO SmartSockets Multicast

In addition to standard publish-subscribe with RTserver and RTclient, SmartSockets provides a multicast feature to further enhance the features and performance of SmartSockets. This option uses reliable multicast, taking full advantage of its bandwidth optimization properties. Multicast is an efficient way of routing a message to multiple recipients. The SmartSockets Multicast feature enables messages to be multicast to RTclients. SmartSockets Multicast uses the PGM protocol to route messages and a new RT process called RTgms to handle the message routing. There are new options for RTclients, and an extended logical connection name that allows the RTclient to connect to the RTgms process. To enable an RTclient to receive or send multicast messages, the RTclient simply connects to the RTgms process, instead of connecting to an RTserver.

To use multicast with SmartSockets, you must have purchased a separate license for the SmartSockets Multicast feature. Contact your TIBCO sales representative for more information. Any RTclients receiving multicast must be running with the SmartSockets Version 6.0 runtime libraries or higher. All RTservers should be at the same SmartSockets version level as the RTgms processes.

When Should I Use Multicast?

The SmartSockets applications that benefit most from multicast are those that distribute data among many network hosts simultaneously, rather than exchanging data with a few hosts at a time. Two models of data distribution exist: one-to-many and many-to-many. There are numerous examples of applications that implement these models.

A one-to-many application sends the same information to many receivers simultaneously. The data flow is one-way, from a single sender. An application that sends time-critical, real-time information, such as stock quotes, is an ideal example of a one-to-many application that benefits from multicast. Another example is a real-time satellite telemetry feed which must be distributed immediately to multiple workstations. Applications that send news and weather are also good examples of applications suitable for multicast. Any application that sends widely distributed information can benefit from multicast, even if the information is not needed as immediately as some of the previous examples. The analogous technical model is broadcast radio and television. The types of content these applications typically send and the schedule coordination requirements are similar.

A many-to-many application is one that shares information with a number of machines simultaneously. In other words, the data flow is bi-directional. Each receiver is also a sender. Each many-to-many application acts as a one-to-many application as it sends data and as a many-to-one application as it receives data.

Many SmartSockets applications involve one-to-many or many-to-many data distribution, where one or multiple sources are publishing the same information (messages) to multiple receivers. Examples can be found across industries:

- Financial: communication of market data to traders, brokers and the general public
- Manufacturing and process control: real-time collection, monitoring and distribution of sensor data from factory floors
- Military or aerospace: real-time signal collection and distribution of data from satellites, radar, and integration and test stands
- Telecommunications: real-time monitoring and control of network devices, and call and service routing
- General: video and audio conferencing for remote meetings and teleconferencing

SmartSockets Multicast efficiently supports this one-to-many message distribution by enabling publishers to send a single copy of a message to multiple recipients who have subscribed to the same subject. This is more efficient than requiring the source to send an individual copy of a message to each subscriber, also called point-to-point unicast. In point-to-point unicast, the number of subscribers is limited by the bandwidth available to the publisher. SmartSockets Multicast is also more efficient than broadcasting one copy of the message to all nodes (broadcast) on the network, because many nodes may not want the message, and because broadcasts, in general, are limited to a single subnet.

SmartSockets Multicast is an efficient way to send messages when those messages are sent to many RTclients all receiving over the same subject on the same network. However, to use the PGM protocol, your network hardware, such as routers and switches, must support multicasting. For more information about what is required for a network that supports multicasting, contact TIBCO Product Support.

For more information on multicast, see Chapter 10, Using Multicast.

Essential API Functions

The SmartSockets publish-subscribe API covers a wide range of function. While all of this function is available to a SmartSockets application developer, the typical SmartSockets application uses only a small fraction of the available functions. The following is a list of the few functions that every SmartSockets application uses, followed by a brief description.

If you plan to use SmartSockets with C, see the *TIBCO SmartSockets Application Programming Interface* reference for the information you need to use these functions. It contains the complete reference information for all the C API functions.

If you plan to use SmartSockets with C++, see the *TIBCO SmartSockets C++ User's Guide* reference for the information you need to use these functions.

If you plan to use SmartSockets with Java, see the *TIBCO SmartSockets Java Library User's Guide and Tutorial* and the online Java reference in JavaDoc format for the information you need. The names of the C functions are similar to the names of the classes and methods used for Java. If you are interested in support for the Java Message Service (JMS), contact TIBCO Product Support for more information on TIBCO JMS products.

Message Type Functions:

These functions create and retrieve information about message types:

TipcMtCreate	Create a new message type.
TipcMtLookup	Look up a message type by name. This is necessary to create a new message.

Message Functions:

These functions construct, manipulate and destroy messages:

<code>TipcMsgAddNamed*</code>	This is a class of functions to add various types of fields to a message by name, including binaries, strings, and reals.
<code>TipcMsgAppend*</code>	This is a class of functions to append various types of fields to a message, including binaries, strings, and reals.
<code>TipcMsgCreate</code>	Create a message.
<code>TipcMsgDeleteNamed</code>	Delete fields to a message by name, including binaries, strings, and reals.
<code>TipcMsgDestroy</code>	Destroy a message.
<code>TipcMsgGetNamed*</code>	This is a class of functions that gets a field by name. Analogous to the <code>TipcMsgAddNamed*</code> functions, there is a "get named" for binaries, strings, reals, and other types.
<code>TipcMsgNext*</code>	This is a class of functions that reads the next field of the type specified. Analogous to the <code>TipcMsgAppend*</code> functions, there is a "get next" for binaries, strings, reals, and other types.
<code>TipcMsgSetCurrent</code>	Set the current field of a message.
<code>TipcMsgSetDest</code>	Set the destination, typically the subject name to which this message will be published.
<code>TipcMsgUpdateNamed*</code>	This is a class of functions to update various types of fields to a message by name, including binaries, strings, and reals.
<code>TipcSrvMsgSend</code>	Publish a message.

Communication Functions:

These are RTclient functions that communicate with RTServer to receive and process messages:

TipcSrvMainLoop	Read and process messages from RTServer.
TipcSrvMsgNext	Get the next message from the connection to RTServer.
TipcSrvMsgProcess	Process a message in the connection to RTServer.
TipcSrvMsgSend	Publish (send) a message through the connection to RTServer.
TipcSrvSubjectCbCreate	Define a new callback to be invoked when a message is received to a particular subject.
TipcSrvSubjectDefaultCbCreate	Define a default subject callback, to be invoked when no other subject callback is defined for that subject.
TipcSrvSubjectSetSubscribe	Start or stop subscribing to a subject.

Utility Functions:

These functions help with startup, debugging and clean up. See the *TIBCO SmartSockets Utilities* reference for the information you need to use these functions:

TutCommandParseFile	Parses commands from a file for startup configuration.
TutErrNumGet	Get the value of the global SmartSockets error number.
TutErrStrGet	Get the SmartSockets global error number as a descriptive string.
TutExit	Call exit handlers and terminate the process. This is required in Windows applications.
TutOut	Unconditionally print out to a SmartSockets output window.

Working With RTclient

This section discusses how to create, access, and destroy connections. Because the focus of this chapter is on the publish-subscribe model, the connections that we describe are usually between the RTclient and an RTserver. To learn more about working with connections in general, see Chapter 2, Connections. The following example programs show the publish-subscribe code used to publish a small data frame of messages, including a user-defined message type, between two RTclients through RTserver. The programs also show how to work with subjects. There are two parts to the example: a sending RTclient and a receiving RTclient. An RTclient can both send messages to and receive messages from RTserver, but this example only shows the two processes doing one or the other. This example only shows a few of the publish-subscribe features built on top of connections.

The source code files for this example are located in the following directory:

UNIX:

```
$RTHOME/examples/smrtsock/manual
```

OpenVMS:

```
RTHOME: [EXAMPLES.SMRTSOCK.MANUAL]
```

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

The online source files have additional `#ifdefs` to provide C++ support; these `#ifdefs` are not shown to simplify the example.

Example 14 Common Header Source Code

```
/* rtclient.h -- common header for RTclient examples */

#define EXAMPLE_PROJECT "example"
#define EXAMPLE_SUBJECT "rcv"
#define EXAMPLE_MT_NAME "example_mt"
#define EXAMPLE_MT_NUM 42
#define EXAMPLE_MT_GRAMMAR "int4 /*code*/ str /*explanation*/"

void create_ud_msg_types();
```

Example 15 Common Utility Source Code

```
/* rtclutil.c -- RTclient example utilities */

#include <rtworks/ipc.h>
#include "rtclient.h"
```

```

/* ===== */
/*..create_ud_msg_types -- create example user-defined msg types */
void create_ud_msg_types()
{
    T_IPC_MT mt;

    /* Create our user-defined message type. */
    mt = TipcMtCreate(EXAMPLE_MT_NAME, EXAMPLE_MT_NUM,
                     EXAMPLE_MT_GRAMMAR);
    if (mt == NULL) {
        TutOut("Could not create example message type: error <%s>.\n",
              TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
    /* Add the new message type to the DATA logging category. */
    if (!TipcSrvLogAddMt(T_IPC_SRV_LOG_DATA, mt)) {
        TutOut("Could not add example mt to DATA category: error
        <%s>.\n",
              TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
} /* create_ud_msg_types */

```

Example 16 Receiver Source Code

```

/* rtclrcv.c -- RTclient example receiver */

/* The receiving RTclient creates its connection to RTserver, */
/* subscribes to subjects, and receives and processes messages. */

#include <rtworks/ipc.h>
#include "rtclient.h"

/* ===== */
/*..cb_process_numeric_data -- process callback for NUMERIC_DATA */
static void T_ENTRY cb_process_numeric_data(
    T_IPC_CONN conn,
    T_IPC_CONN_PROCESS_CB_DATA data,
    T_CB_ARG arg)
{
    T_STR name;
    T_REAL8 value;

    TutOut("Entering cb_process_numeric_data.\n");

    /* set current field to first field in message */
    if (!TipcMsgSetCurrent(data->msg, 0)) {
        TutOut("Could not set current field of message: error <%s>.\n",
              TutErrStrGet());
        return;
    }
}

```

```

    /* access and print fields */
    while (TipcMsgNextStrReal8(data->msg, &name, &value)) {
        TutOut("%s = %s\n", name, TutRealToStr(value));
    }
    /* make sure we reached the end of the message */
    if (TutErrNumGet() != T_ERR_MSG_EOM) {
        TutOut("Did not reach end of message: error <%s>.\n",
            TutErrStrGet());
    }
} /* cb_process_numeric_data */

/* ===== */
/*..cb_default -- default callback */
static void T_ENTRY cb_default(
    T_IPC_CONN conn,
    T_IPC_CONN_DEFAULT_CB_DATA data,
    T_CB_ARG arg)
{
    T_IPC_MT mt;
    T_STR name;

    /* This callback function will be called for messages where */
    /* there are no process callbacks for that message type. */
    TutOut("Entering cb_default.\n");

    /* print out the name of the type of the message */
    if (!TipcMsgGetType(data->msg, &mt)) {
        TutOut("Could not get message type from message: error
<%s>.\n",
            TutErrStrGet());
        return;
    }
    if (!TipcMtGetName(mt, &name)) {
        TutOut("Could not get name from message type: error <%s>.\n",
            TutErrStrGet());
        return;
    }
    TutOut("Message type name is %s.\n", name);
} /* cb_default */
/* ===== */
/*..main -- main program */
int main(argc, argv)
int argc;
char **argv;
{
    T_OPTION option;
    T_IPC_MT mt; /* message type for creating callbacks */

    /* Create user-defined message types. */
    create_ud_msg_types();

```

```

/* Set the option Project to partition ourself. */
option = TutOptionLookup("project");
if (option == NULL) {
    TutOut("Could not look up option named project: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TutOptionSetEnum(option, EXAMPLE_PROJECT)) {
    TutOut("Could not set option named project: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* Allow a command-line argument containing the name of a */
/* SmartSockets startup command file. This file can be used */
/* to set options like Server_Names. */
if (argc == 2) {
    if (!TutCommandParseFile(argv[1])) {
        TutOut("Could not parse startup command file %s: error
<%s>.\n",
            argv[1], TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
}
else if (argc != 1) { /* too many command-line arguments */
    TutOut("Usage: %s [ command_file_name ]\n", argv[0]);
    TutExit(T_EXIT_FAILURE);
}

TutOut("Creating connection to RTserver.\n");
if (!TipcSrvCreate(T_IPC_SRV_CONN_FULL)) {
    TutOut("Could not create connection to RTserver: error
<%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* create callbacks to be executed when certain operations occur */
TutOut("Create callbacks.\n");

/* process callback for NUMERIC_DATA */
mt = TipcMtLookupByNum(T_MT_NUMERIC_DATA);
if (mt == NULL) {
    TutOut("Could not look up NUMERIC_DATA msg type: error
<%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (TipcSrvProcessCbCreate(mt, cb_process_numeric_data, NULL)
    == NULL) {
    TutOut("Could not create NUMERIC_DATA process cb: error
<%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

```

```

/* default callback */
if (TipcSrvDefaultCbCreate(cb_default, NULL) == NULL) {
    TutOut("Could not create default cb: error <%s>.\n",
        TutErrStrGet());
}

TutOut("Start subscribing to standard subjects.\n");
if (!TipcSrvStdSubjectSetSubscribe(TRUE, FALSE)) {
    TutOut("Could not subscribe to standard subjects: error
<%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

TutOut("Start subscribing to the %s subject.\n",
EXAMPLE_SUBJECT);
if (!TipcSrvSubjectSetSubscribe(EXAMPLE_SUBJECT, TRUE)) {
    TutOut("Could not start subscribing to %s subject: error
<%s>.\n",
        EXAMPLE_SUBJECT, TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* If an error occurs, then TipcSrvMainLoop will restart RTserver */
/* and return FALSE. We can safely continue. */
for (;;) {
    if (!TipcSrvMainLoop(T_TIMEOUT_FOREVER)) {
        TutOut("TipcSrvMainLoop failed: error <%s>.\n",
            TutErrStrGet());
    }
}

/* This line should not be reached. */
TutOut("This line should not be reached!!!\n");
return T_EXIT_FAILURE;
} /* main */

```

Example 17 Sender Source Code

```

/
* rtclsnd.c -- RTclient example sender */

/*
This sending RTclient creates its connection and publishes a data frame of messages to a subject
(through RTserver).
*/

#include <rtworks/ipc.h>
#include "rtclient.h"

```

```

/* ===== */
/*..main -- main program */
int main(argc, argv)
int argc;
char **argv;
{
    T_OPTION option;
    T_IPC_MT mt;
    /* Create user-defined message types. */
    create_ud_msg_types();

    /* Set the option Project to partition ourself. */
    option = TutOptionLookup("project");
    if (option == NULL) {
        TutOut("Could not look up option named project: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
    if (!TutOptionSetEnum(option, EXAMPLE_PROJECT)) {
        TutOut("Could not set option named project: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    /* Log outgoing data messages to a message file. Another */
    /* way to set options is to use TutCommandParseStr. */
    TutCommandParseStr("setopt log_out_data log_out.msg");

    /* Allow a command-line argument containing the name of a */
    /* SmartSockets startup command file. This file can be used */
    /* to set options like Server_Names. */
    if (argc == 2) {
        if (!TutCommandParseFile(argv[1])) {
            TutOut("Could not parse startup command file %s: error
<%s>.\n",
                argv[1], TutErrStrGet());
            TutExit(T_EXIT_FAILURE);
        }
    }
    else if (argc != 1) { /* too many command-line arguments */
        TutOut("Usage: %s [ command_file_name ]\n", argv[0]);
        TutExit(T_EXIT_FAILURE);
    }

    TutOut("Creating connection to RTserver.\n");
    if (!TipcSrvCreate(T_IPC_SRV_CONN_FULL)) {
        TutOut("Could not create connection to RTserver: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
}

```

```

TutOut("Publish a frame of data to the receiver's subject.\n");

TutOut("Publishing a TIME message.\n");
mt = TipcMtLookupByNum(T_MT_TIME);
if (mt == NULL) {
    TutOut("Could not look up TIME msg type: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcSrvMsgWrite(EXAMPLE_SUBJECT, mt, TRUE,
                     T_IPC_FT_REAL8, 1.0,
                     NULL)) {
    TutOut("Could not publish TIME message: error <%s>.\n",
           TutErrStrGet());
}

TutOut("Publishing a NUMERIC_DATA message.\n");
mt = TipcMtLookupByNum(T_MT_NUMERIC_DATA);
if (mt == NULL) {
    TutOut("Could not look up NUMERIC_DATA msg type: error
<%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcSrvMsgWrite(EXAMPLE_SUBJECT, mt, TRUE,
                     T_IPC_FT_STR, "voltage",
                     T_IPC_FT_REAL8, 33.4534,
                     T_IPC_FT_STR, "switch_pos",
                     T_IPC_FT_REAL8, 0.0,
                     NULL)) {
    TutOut("Could not publish NUMERIC_DATA message: error <%s>.\n",
           TutErrStrGet());
}

TutOut("Publishing an EXAMPLE message.\n");
mt = TipcMtLookupByNum(EXAMPLE_MT_NUM);
if (mt == NULL) {
    TutOut("Could not look up EXAMPLE msg type: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcSrvMsgWrite(EXAMPLE_SUBJECT, mt, TRUE,
                     T_IPC_FT_INT4, 7,
                     T_IPC_FT_STR, "Seven is your lucky number",
                     NULL)) {
    TutOut("Could not publish EXAMPLE message: error <%s>.\n",
           TutErrStrGet());
}

TutOut("Publishing an END_OF_FRAME message.\n");
mt = TipcMtLookupByNum(T_MT_END_OF_FRAME);
if (mt == NULL) {
    TutOut("Could not look up END_OF_FRAME msg type: error
<%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

```

```

    if (!TipcSrvMsgWrite(EXAMPLE_SUBJECT, mt, TRUE, NULL)) {
        TutOut("Could not publish END_OF_FRAME message: error <%s>.\n",
            TutErrStrGet());
    }

    /* Each RTclient automatically creates a connection process */
    /* callback for CONTROL messages. Use this to send the command */
    /* "quit force" to the receiver's command interface. */
    TutOut("Publishing a CONTROL message to stop the
receiver(s).\n");
    mt = TipcMtLookupByNum(T_MT_CONTROL);
    if (mt == NULL) {
        TutOut("Could not look up CONTROL msg type: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
    if (!TipcSrvMsgWrite(EXAMPLE_SUBJECT, mt, TRUE,
        T_IPC_FT_STR, "quit force",
        NULL)) {
        TutOut("Could not publish CONTROL message: error <%s>.\n",
            TutErrStrGet());
    }

    /* Flush the buffered outgoing messages to RTserver. */
    if (!TipcSrvFlush()) {
        TutOut("Could not flush messages to RTserver: error <%s>.\n",
            TutErrStrGet());
    }

    /* Completely disconnect from RTserver. */
    if (!TipcSrvDestroy(T_IPC_SRV_CONN_NONE)) {
        TutOut("Could not destroy connection to RTserver: error
<%s>.\n",
            TutErrStrGet());
    }

    TutOut("Sender RTclient exiting successfully.\n");
    return T_EXIT_SUCCESS; /* all done */
} /* main */

```


Compiling, Linking, and Running

To compile, link, and run the example programs, first you must either copy the programs to your own directory or have write permission in the following directory or partitioned datasets (MVS):

UNIX:

```
$RTHOME/examples/smrtsock/manual
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]
```

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

Step 1 **To compile and link the programs, use:**

UNIX:

```
$ rtlink -o rtclrcv.x rtclrcv.c rtclutil.c
$ rtlink -o rtclsnd.x rtclsnd.c rtclutil.c
```

OpenVMS:

```
$ cc rtclrcv.c, rtclsnd.c, rtclutil.c
$ rtlink /exec=rtclrcv.exe rtclrcv.obj, rtclutil.obj
$ rtlink /exec=rtclsnd.exe rtclsnd.obj, rtclutil.obj
```

Windows:

```
$ nmake /f clrvw32m.mak
$ nmake /f clsnw32m.mak
```

On UNIX the `rtlink` command by default uses the `cc` command to compile and link. To use a C++ compiler or a C compiler with a name other than `cc`, set the environment variable `CC` to the name of the compiler, and `rtlink` then uses this compiler. For example, these commands could be used to compile and link on UNIX with the GNU C++ compiler `g++`:

```
$ env CC=g++ rtlink -o rtclrcv.x rtclrcv.c rtclutil.c
$ env CC=g++ rtlink -o rtclsnd.x rtclsnd.c rtclutil.c
```

Step 2 Start RTserver

Before running the programs, start RTserver. An RTclient will not by default automatically start an RTserver. You can change this behavior by using a start prefix in RTclient's Server_Names option. For more information, see Start Prefix on page 195 and Starting and Stopping RTserver on page 284.

Use this command to start RTserver on 32-bit platforms:

```
$ rtserver
```

To run the programs, start the receiving process first in one terminal emulator window and then the sending process in another terminal emulator window.



On platforms that support both 32- and 64-bit, use the `rtserver64` command to run the 64-bit version of RTserver.

Step 3 Start the receiving program in the first window**UNIX:**

```
$ rtclrcv.x
```

OpenVMS:

```
$ run rtclrcv.exe
```

Windows:

```
$ rtclrcv
```

Step 4 Start the sending program in the second window**UNIX:**

```
$ rtclsnd.x
```

OpenVMS:

```
$ run rtclsnd.exe
```

Windows:

```
$ rtclsnd.exe
```

Here is an example of the receiving process output:

```

Creating connection to RTserver.
Connecting to project <example> on <_node> RTserver
Using local protocol
Message from RTserver: Connection established.
Start subscribing to subject </_node_5415>
Create callbacks.
Start subscribing to standard subjects.
Start subscribing to subject </_node>
Start subscribing to subject </_all>
Start subscribing to the rcv subject.
Entering cb_default.
Message type name is time.
Entering cb_process_numeric_data.
voltage = 33.4534
switch_pos = 0
Entering cb_default.
Message type name is example_mt.
Entering cb_default.
Message type name is end_of_frame.

```

Here is an example of the sending process output:

```

Now logging outgoing data-related messages to <log_out.msg>
Creating connection to RTserver.
Connecting to project <example> on <_node> RTserver
Using local protocol
Message from RTserver: Connection established.
Start subscribing to subject </_node_5415>
Publish a frame of data to the receiver's subject.
Publishing a TIME message.
Publishing a NUMERIC_DATA message.
Publishing an EXAMPLE message.
Publishing an END_OF_FRAME message.
Publishing a CONTROL message to stop the receiver(s).
Now logging outgoing data - related messages to <log_out.msg>
Sender RTclient exiting successfully.

```

The message file `log_out.msg`, which is created by the sender:

```

time rcv 1
numeric_data rcv {
  voltage 33.4534
  switch_pos 0
}
example_mt rcv 7 "Seven is your lucky number"
end_of_frame rcv
control rcv "quit force"

```

More than one receiving process can be started if desired. Each receiving process receives the same messages from the sending process. To run three receivers, start the receiving program in three separate windows or in batch (background), using the same command as shown in Step 3.

Include Files

Code written in C or C++ that uses the SmartSockets Application Programming Interface (API) must include the header file `<rtworks/ipc.h>`. This file is located in these directories or partitioned datasets (MVS):

UNIX:
`$RTHOME/include/$RTARCH/rtworks`

OpenVMS:
`RTHOME:[INCLUDE.RTWORKS]`

Windows:
`%RTHOME%\include\rtworks`

The SmartSockets IPC API includes all the functions used for interprocess communication.

Differences Between the TipcConn* and TipcSrv* API

Most TipcConn* connection functions have an equivalent TipcSrv* function that operates only on the connection to RTserver. For most TipcConn* functions that have a TipcSrv* equivalent, the calling sequences are identical except that the T_IPC_CONN first parameter in the TipcConn* function disappears in the TipcSrv* function because the connection is always with RTserver. For example, the C/C++ function prototype for TipcConnMsgProcess is:

```
T_BOOL TipcConnMsgProcess(T_IPC_CONN conn, T_IPC_MSG msg);
```

The corresponding function prototype for TipcSrvMsgProcess is:

```
T_BOOL TipcSrvMsgProcess(T_IPC_MSG msg);
```

Table 6 shows the TipcSrv* functions that have names similar to TipcConn* functions, but that have different behavior.

Table 6 TipcSrv* Functions With Different Behavior

Function Name	Difference from Related TipcConn* Function
TipcSrvCreate	TipcConnCreate creates an empty connection, while TipcSrvCreate uses several options to find, possibly start, and connect to RTserver.
TipcSrvDestroy	TipcConnDestroy deallocates memory and closes a socket, while TipcSrvDestroy may leave a warm connection to RTserver.

Table 6 *TipcSrv* Functions With Different Behavior (Cont'd)*

Function Name	Difference from Related TipcConn* Function
TipcSrvMsgSend TipcSrvMsgWrite TipcSrvMsgWriteVa	These functions set the sender property of a message to the value in the option <code>Unique_Subject</code> and have an additional <code>check_server_msg_send</code> parameter (of type <code>T_BOOL</code>). <code>TipcSrvMsgWrite</code> and <code>TipcSrvMsgWriteVa</code> also have an additional <code>destination</code> parameter (of type <code>T_STR</code>).
TipcSrvGmdFileDelete	Because <code>TipcSrvCreate</code> calls <code>TipcSrvGmdResend</code> (which will open any existing GMD files and thus not allow them to be deleted), <code>TipcSrvGmdFileDelete</code> does not call <code>TipcConnGmdFileDelete</code> but does use the same algorithm.

Table 7 shows the `TipcConn*` functions that do not have `TipcSrv*` equivalents.

Table 7 *TipcConn* Functions Without TipcSrv* Equivalents*

Function Name	Explanation
TipcConnCreateClient TipcConnCreateServer	An RTclient uses the function <code>TipcSrvCreate</code> , which uses the options <code>Server_Names</code> and <code>Default_Protocols</code> to determine which IPC protocol to use to create a client connection to RTserver. An RTclient always creates a client connection, not a server connection, when it connects to RTserver.
TipcConnAccept	RTserver accepts a connection from RTclient, not the other way around.
TipcConnEncodeCbCreate TipcConnEncodeCbLookup TipcConnDecodeCbCreate TipcConnDecodeCbLookup	RTclient does not support encode and decode callbacks on the connection to RTserver.

Table 8 shows the TipcSrv* functions that do not have TipcConn* equivalents.

Table 8 TipcSrv Functions Without TipcConn* Equivalents*

Functions	Explanation
TipcSrvLogAddMt TipcSrvLogRemoveMt	These functions manipulate standard message file logging types, which do not exist in connections, only in RTclient.
TipcSrvCreateCbCreate TipcSrvCreateCbLookup TipcSrvDestroyCbCreate TipcSrvDestroyCbLookup TipcSrvSubjectCbCreate TipcSrvSubjectCbDestroyAll TipcSrvSubjectCbLookup TipcSrvSubjectDefaultCbCreate TipcSrvSubjectDefaultCbLookup TipcSrvTraverseCbCreate TipcSrvTraverseCbLookup	These functions manipulate server create callbacks, server destroy callbacks, server names traverse callbacks, and subject callbacks, which do not exist in connections, only in RTclient.
TipcSrvSubjectGetSubscribe TipcSrvSubjectSetSubscribe TipcSrvSubjectGetSubscribeLb TipcSrvSubjectSetSubscribeLb TipcSrvSubjectGmdInit TipcSrvSubjectLbInit TipcSrvSubjectTraverseSubscribe	These functions manipulate subjects, which do not exist in connections, only in RTserver and RTclient.
TipcSrvStdSubjectSetSubscribe TipcSrvStdSubjectTraverse	These functions manipulate standard subjects, which do not exist in connections, only in RTserver and RTclient.
TipcSrvGetConnStatus TipcSrvIsRunning TipcSrvStop	Miscellaneous RTclient utility functions.
TipcSrvGmdMsgServerDelete TipcSrvGmdMsgStatus	These functions provide GMD that exists in RTclient but not in connections.

Setting Options

Options allow you to customize your RTclient's configuration. A user-defined RTclient does not have any standard startup command files. It is up to you to decide which command files, if any, should be loaded. Startup command files can be loaded with the function `TutCommandParseStr` or by using `TutCommandParseFile`. Options can also be set using the function `TutOptionSetType`, where *Type* is replaced with the type of the option. All RTclient options are discussed in detail in Chapter 8, Options Reference.

Creating a Connection to RTserver

After an RTclient has initialized its options, a connection to RTserver can be created. There are two kinds of global connections to RTserver: a warm connection and a full connection. Warm connections are discussed in Warm Connection to RTserver on page 235. Through the remainder of this document, the term connection to RTserver is used to mean a full global connection to RTserver.

The function `TipcSrvCreate` is used to create a connection to RTserver. For example:

```
if (!TipcSrvCreate(T_IPC_SRV_CONN_FULL)) {
    TutOut("Could not create connection to RTserver: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

The `connect` command provides a way to use `TipcSrvCreate` from the SmartSockets command interface (see the command reference for `connect`, page 595).

An RTclient cannot have a global connection to more than one RTserver at a time, although it can move its connection from one RTserver to another RTserver at any time. However, instead of a single global connection, an RTclient can have multiple RTserver connections using a special type of multiple connection. For more information in C, see [Connecting to Multiple RTservers](#), page 248. For more information in Java, see the *TIBCO SmartSockets Java Library User's Guide and Tutorial*.

TipcSrvCreate uses several options to control the creation of the connection to RTserver:

Default_Protocols	specifies a list of IPC protocols to try if no protocol is specified.
Server_Disconnect_Mode	specifies the action RTserver should take when RTclient disconnects.
Server_Names	specifies a list of logical connection names used to find and start RTserver.
Server_Start_Delay	specifies the number of seconds to sleep between traversals of the Server_Names option.
Server_Start_Max_Tries	specifies the number of times to traverse the Server_Names option.
Server_Start_Timeout	specifies the number of seconds to wait for RTserver to initialize.
Udp_Broadcast_Timeout	specifies the number of seconds to wait for broadcast replies.

The options Default_Subject_Prefix, Project, Server_Disconnect_Mode, and Unique_Subject do not directly control connecting to RTserver, but they are used once RTclient has found an RTserver.

Creating a Connection to RTgms

If the SmartSockets system is enabled for multicast and the RTclient wants to use multicast, the RTclient must connect to an RTgms instead of connecting to an RTserver. In most cases, the only change required is to the Group_Names and Server_Names options in the RTclient command (.cm) file.

The Group_Names option specifies which multicast group the RTclient belongs to. The default is `rtworks`, and you only need to change the value if you are not using that group name.

The Server_Names option must provide the logical connection name for an RTgms process instead of the logical connection name for an RTserver process.

For example, your RTclient command file might contain:

```
setopt group_names rtworks
setopt server_names tcp:nodea
```

Let's assume the RTclient should belong to the multicast group `mcast1`, and should connect to the RTgms on `nodea` using the default port, which is 5104. Change the lines to:

```
setopt group_names mcast1
setopt server_names pgm:nodea
```


If you want to connect to an RTgms that is not using the default port, change the `Server_Names` line to:

```
setopt server_names pgm:nodea:tcp.6000
```

which connects to the RTgms on `nodea` using port 6000. For more information on the format of RTgms addresses, see [Address for Multicast](#), page 194.

After ensuring your options are set correctly for multicast, use the function `TipcSrvCreate` to connect to the RTgms process the same way you connect to an RTserver process:

```
if (!TipcSrvCreate(T_IPC_SRV_CONN_FULL)) {
    TutOut("Could not create connection to RTgms: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

When connecting to an RTgms process, you cannot specify `pgm:localhost` alone. You must also include TCP as the protocol:

```
pgm:localhost:tcp
```

In addition to the options used in creating connections for an RTclient, there are special options that apply only to multicast connections. The names of these options begin with `Pgm_`. These options must be set in a multicast command file, `mcast.cm`. For information on these options and on the multicast command file, see [Chapter 10, Using Multicast](#). If your SmartSockets system does not have the multicast option installed, you receive an error when you attempt to connect to RTgms.

Belonging to a Project

When an RTclient creates a connection to RTserver, it becomes part of the project named in the option `Project`. As described in [Projects](#) on page 156, each project is self-contained, and no messages can be sent between projects.

Logical Connection Names for RT Processes

SmartSockets RT processes, such as RTserver and RTclient, simplify the creation of connections with logical connection names that are specified consistently for all protocols. RTserver and RTclient take full advantage of the logical connection name feature of connections. For an introduction to these features, see the section Logical Connection Names, page 101. RTserver uses logical connection names to create its connections, and RTclient uses logical connection names to create a client connection to an RTserver or RTgms.

The most important option for connecting to another RT process is `Server_Names`, which is used to find and start an RT process such as RTserver. `Server_Names` is a list of logical connection names. Each logical connection name has the form:

protocol:node:address

which can be shortened to *protocol:node*, *protocol*, or simply *node* for normal connections. The `TipcSrvCreate` function expands the shortened forms to the long *protocol:node:address* form and uses default values for the parts that are not specified. For RTclient to find an RTserver, one of the logical connection names used by RTclient must exactly match one of the logical connection names used by that RTserver (for example, the name `tcp:abc:1234` does not match the name `tcp:xyz:1234`).

For RTclient to automatically start or re-start an RTserver, `Server_Names` must use one of the start prefixes listed in Start Prefix on page 195. By default, the start prefix is `start_never`, meaning that RTclient cannot start RTserver.

Protocol Portion

The *protocol* part of the connection name refers to an IPC protocol type. Examples of common protocols are `tcp`, `local`. There is also the special SmartSockets protocol defined for multicast, `pgm`. The `pgm` protocol is only supported if your SmartSockets system has the multicast option installed.

If *protocol* is omitted from the connection name, then all protocols listed in the `Default_Protocols` option are tried in order. These protocol names (with the exception of `udp_broadcast`, which is described in the next section) map to the `TipcConnCreateClient` function described in the section Creating a Client Connection, page 104.

If you specify `pgm` for multicast as the protocol, the address portion of the logical connection name uses a different format than for other protocols. See Address for Multicast, page 194.

The Udp_Broadcast Protocol

The `udp_broadcast` protocol is only used to find RTserver, and not to start RTserver or send or receive messages. When the `udp_broadcast` protocol is used, a packet is broadcast to all nodes on the local network to attempt to find an RTserver. If an RTserver receives the broadcast packet, it responds to the request with a list of logical connection names for RTclient to use to find that RTserver. RTclient waits the number of seconds specified in the `Udp_Broadcast_Timeout` option for RTserver to respond to its broadcast, and then connects to the RTserver that responds first.

There are several categories of UDP broadcast addresses that vary in how wide an area they are distributed over. The book *TCP/IP Illustrated, Volume 1: The Protocols* by W. Richard Stevens has an excellent discussion of broadcasting. The categories are shown below.

- the limited broadcast address is 255.255.255.255 and is intended for a single network interface, but may work differently on a node with multiple network interfaces
- the net-directed broadcast address is intended for a non-subnetted network
- the subnet-directed broadcast address is intended for a single TCP/IP subnet
- the all-subnets-directed broadcast address is intended for all subnets

Unfortunately, these categories behave differently depending on the operating system and network router configuration used, making it hard to pick a default broadcast address that works in all situations. The limited and all-subnets-directed broadcast addresses do not work on several operating systems supported by SmartSockets.

Therefore, by default the broadcast packet is sent to the subnet-directed address of the main network interface. In non-subnetted networks, the net-directed, subnet-directed, and all-subnet-directed broadcast addresses are equivalent, which makes the subnet-directed address the most reasonable default for most networks.

When using the `udp_broadcast` protocol, a dotted-decimal address can be used for the node portion of the logical connection name to override the broadcast default. For example, the logical connection name `udp_broadcast:255.255.255.255` can be used on many platforms as a limited broadcast address.



SmartSockets does not support broadcast over several network interfaces at the same time.

Node Portion

The *node* part of the connection name refers to a computer node name. If *node* is omitted from the connection name, then `_node`, the current node, is used as a default. Using the default node works if your RTserver or RTgms process to which you are connecting is on the same node as your RTclient. If you configured them to be on a different node, you must specify that node for the *node* portion of the connection name.

Address Portion

The *address* part of the connection name refers to a protocol-specific IPC location, such as a `tcp` port number. If *address* is omitted from the connection name, a protocol-specific default *address* is used. The default addresses for all protocols are:

Protocol Name	Default Address
local	RTSERVER
pgm	<i>unicast_protocol.address</i> (See Address for Multicast.)
tcp	5101
udp_broadcast	5101

Address for Multicast

If you are connecting to an RTgms process for multicast, instead of to an RTserver, the address portion of your logical connection name is a different format than for other protocols. The format for multicast is:

unicast_protocol.address

where:

unicast_protocol specifies the unicast protocol to use when sending data to an RTgms. The valid values you can specify are `tcp` or `local`.

This field is optional, unless you specified `localhost` for the node on a UNIX system. If you specify `localhost` for the node, the unicast protocol must be `tcp`.

On Windows, the default is `tcp`.

On UNIX, where the default is `local`, you must specify `tcp` as the unicast protocol:

`pgm:localhost:tcp`

address specifies the address portion of the unicast logical connection name used by the RTgms to receive data. This is the address or port defined for the RTgms. The default is 5104.

This field is optional.

If you specify a multicast format address, and your SmartSockets system does not have the multicast option installed, you receive an error when you attempt to connect to RTgms.

Start Prefix

Each logical connection name in RTclient can also have a logical connection name modifier called a start prefix at the front, which must be separated from the name with a colon:

start_prefix: protocol: node: address

For more information, see Logical Connection Name Modifiers on page 291.

The start prefix controls if and when an RTclient tries to start RTserver. On Windows, if an RTserver has been installed as a Windows service, you must also set the proper environment variable before an RTclient can use a start prefix to start that RTserver. Set the RTSERVER_CMD environment variable to:

```
net start "SmartSockets RTserver"
```

The valid start prefixes are:

- start_always* RTclient always tries to start RTserver if it cannot create the connection to RTserver.
- start_on_demand* RTclient only tries to start RTserver if the RTclient has tried all names in *Server_Names* at least once. This is useful for only starting RTserver if an existing one cannot be found. For this start prefix to be used, the *Server_Start_Max_Tries* option must be increased above the default value of 1 (see pseudo code in Finding and Starting RTserver on page 197 for details).
- start_never* RTclient never tries to start RTserver. This is the default.

If no start prefix is specified, an implicit default start prefix of *start_never* is used. If the *protocol* portion of the logical connection name is *local* and the *node* portion is not the name of the current node, an implicit start prefix of *start_never* is used because the local protocol cannot connect to a remote RTserver.



The start prefix is only for an RTclient to start an RTserver. It cannot be used by any other process to start an RTserver or used by an RTclient to start a process other than an RTserver.

Randomizing Server_Names

The special value `_random` can be used in the `Server_Names` option in an `RTclient` to randomize the list of `RTserver` names. Every name occurring after `_random` is tried in a random order. This enables load balancing of the `RTclient` connections to the associated `RTservers`. For example, if the option `Server_Names` is set to `workstation1, _random, workstation2, workstation3, workstation4`, then `workstation1` is always tried first, and then `workstation2`, `workstation3`, and `workstation4` are tried in a random order each call to the `TipcSrvCreate` function.

Scanning Server_Names

The special value `_next` can be used in the `server_names` option for an `RTclient`. This special value changes the starting point at which the `RTclient` traverses the `server_names` option in the event of server failure. If included, the `_next` value must be the first value presented in the `server_names` list. Each server name is tried in sequential order, starting with the "next" server listed after the currently connected server. This prevents the `RTclient` from attempting to reconnect to a server to which it had previously lost its connection.

For example:

```
setopt server_names _next, server1, server2, server3, server4
```

When the `RTclient` initially tries to make a connection to a server, it attempts to connect to `server1`. Once a connection is established to `server1`, should a failure occur in `server1`, the `RTclient` attempts reconnection by starting with the "next" server in the list, `server2`. Without the `_next` property, the `RTclient` would always attempt to connect to the list of servers starting with the first server in the list. The `_next` option does not eliminate any servers from the list; if the `RTclient` reaches the end of the list when using the `_next` option, then the `RTclient` circles back around to the front of the list and continues connection attempts until all servers in the list have been attempted.

RTservers with Multiple IP Addresses

Generally, for an RTclient to find an RTserver, the logical connection name used in `Server_Names` by the RTclient must exactly match the logical connection name used in `Conn_Names` by the RTserver. For example, the name `tcp:moe:1234` does not match the name `tcp:conan:1234`. However, this perfect match is not needed when the RTclient is trying to find an RT server that is listening on all IP addresses on a machine with more than one IP address, that is, multi-homed.

An RTserver listens on all IP addresses of a machine if the *node* portion of its logical connection name (specified in `Conn_Names`) is `_any`. Using `_node` for *node* causes the RTserver to listen only on the default IP address. For example, if an RTserver is on a machine with IP addresses `ipaddr_a` and `ipaddr_b`, and its `Conn_Names` is set to `tcp:_any:5101`, an RTclient can connect to this RTserver if its `Server_Names` option is set to either `tcp:ipaddr_a:5101` or `tcp:ipaddr_b:5101`.

If you want the RTserver to listen only on a specific IP address, you can specify that IP address in the `Conn_Names` option. However, there is a known issue that occurs if running an RTserver on a machine with multiple IP addresses when using `udp_broadcast` to discover RTservers. Any responding RTserver replies with only its default IP address, even if its `Conn_Names` option is specifically set to listen on an IP address other than the default. If you are using `udp_broadcast`, be sure to include the default IP address in the RTserver `Conn_Names` option, or use `_node` or `_any` in the `Conn_Names` option.



Using the keyword `_any` in `Conn_Names` is discouraged for RTserver to RTserver connections. When an RTserver connects to another RTserver whose `Conn_Names` use `_any`, the RTserver might attempt to reconnect every `Server_Reconnect_Interval` seconds. This is a known problem and will be fixed in a future release.

Finding and Starting RTserver

An RTclient traverses the list of names in `Server_Names` at most `Server_Start_Max_Tries`. Between each traversal RTclient sleeps for `Server_Start_Delay` seconds. Each time an RTclient tries to start RTserver, it waits for up to `Server_Start_Timeout` seconds for that RTserver to finish initializing. For each entry in `Server_Names`, RTclient tries to connect, then possibly tries to start RTserver (depending on what start prefix is used), then tries to connect again.

There is a server names traverse callback available to RTclients. This callback is executed before each attempt to connect to an RTserver in the `Server_Names` list. For more information, see *Server Names Traverse Callbacks*, page 239.

This pseudo code shows the algorithm used by TipcSrvCreate:

```
randomize Server_Names list if _random is used

for (each $try from 1 to Server_Start_Max_Tries) {

    for (each $name in Server_Names) {
        connect to RTserver using $name
        if (connection could be created) {
            if (no connection previously existed) {
                resend old GMD messages
            }
            execute server create callbacks
            return
        }
    }
    else {
        if ($name has the start prefix "start_always"
            or $name has no start prefix
            or ($name has the start prefix "start_on_demand"
                and $try > 1)) {
            start RTserver with command "rtserver -node node"
            wait up to Server_Start_Timeout seconds for RTserver to init
            connect to RTserver using $name
            if (connection could be created) {
                if (no connection previously existed) {
                    resend old GMD messages
                }
                execute server create callbacks
                return
            }
        }
        } /* if we should try to start RTserver */
    } /* if no connection */
} /* for each name */

sleep for Server_Start_Delay seconds
} /* for each try */
```



The start_on_demand start prefix only starts RTserver if the option Server_Start_Max_Tries is set to a value larger than 1 (the default).

If TipcSrvCreate is able to successfully create a connection, it calls the server create callbacks (see RTclient-Specific Callbacks on page 237 for information on server create callbacks).

The options shown in the example are only accessed when the RTclient creates a connection to RTserver. If one of these options is changed, it does not take effect until the next time the RTclient creates a connection to RTserver. See Changing RTclient Options on page 247 for an example of how to change these options and have them take effect immediately.

Starting RTserver on a Remote Node

When RTclient starts RTserver, it uses the *node* portion of the logical connection name to determine which node to attempt to start RTserver on. To start RTserver on a remote node, it must be possible to perform a remote shell *rsh* (*remsh* on HP-UX) to that node without a password. If *rsh* to the remote system does not work, then the RTclient is not able to start RTserver. Use this command to test *rsh* (on all platforms except HP-UX):

```
$ rsh node rtserver -help
```

On HP-UX, use this command:

```
$ remsh node rtserver -help
```



On platforms that support both 32- and 64-bit, use the *rtserver64* command to run the 64-bit version of RTserver.

If a prompt for a password appears when this command is entered, contact the system administrator for help in configuring the remote shell (*rsh* or *remsh*) to not require a password.

Starting RTserver on a remote machine under OpenVMS requires a TCP/IP package running with support for the *rsh* command, such as MultiNet.

Local Protocol Failure

Occasionally, you may find that RTclient can no longer connect to RTserver using the *local* protocol, but can still connect using the *TCP* protocol. For example:

```
Connecting to project <rtworks> on <_node> RTserver.
Using local protocol.
connect: No such file or directory
Could not connect to <_node> RTserver.
Connecting to project <rtworks> on <_node> RTserver.
Using tcp protocol.
Message from RTserver: Connection established.
```

One possible scenario is the *local* socket file used by RTserver (stored in the directory specified by the function *TutGetSocketDir*) gets deleted somehow, perhaps by an overzealous system administrator. To fix the problem, simply restart RTserver, and a new socket file is created.

Automatically Connecting to RTserver

The above example programs explicitly call the function `TipcSrvCreate` to create a connection to `RTserver`. `RTclient` can also automatically create a connection to `RTserver` if one is needed, such as when `RTclient` tries to publish a message to `RTserver` before it has created a connection to `RTserver`. Most of the `TipcSrv*` functions automatically attempt to create a connection to `RTserver` (by calling `TipcSrvCreate` themselves) if one is needed and the option `Server_Auto_Connect` is set to `TRUE`.

The standard `SmartSockets` modules also use `Server_Auto_Connect` in a second way. During initialization, each of these standard processes uses `Server_Auto_Connect` to determine whether or not to require a connection to `RTserver`. If `Server_Auto_Connect` is `TRUE`, the process requires a connection to `RTserver`, and creates one, if necessary, to start up. To run the standard processes without any connection to `RTserver`, `Server_Auto_Connect` must be set to `FALSE`. Some standard data sources in the standard `SmartSockets` processes also require a connection to `RTserver` regardless of the value of `Server_Auto_Connect`.

Table 9 shows the `TipcSrv*` functions that do not automatically create a connection to `RTserver`.

Table 9 *TipcSrv* Functions That do not Automatically Create a Connection to RTserver*

Function Name	Explanation
<code>TipcSrvLogAddMt</code> <code>TipcSrvLogRemoveMt</code>	The standard message file logging types can only be manipulated before a connection to <code>RTserver</code> is created.
<code>TipcSrvCreateCbCreate</code> <code>TipcSrvCreateCbLookup</code> <code>TipcSrvDestroyCbCreate</code> <code>TipcSrvDestroyCbLookup</code>	Manipulating server create callbacks and server destroy callbacks does not need a connection to <code>RTserver</code> .
<code>TipcSrvCreate</code> <code>TipcSrvDestroy</code>	These functions explicitly manipulate the connection to <code>RTserver</code> .
<code>TipcSrvGetConnStatus</code> <code>TipcSrvIsRunning</code> <code>TipcSrvStop</code>	Miscellaneous <code>RTclient</code> utility functions.
<code>TipcSrvGmdFileDelete</code>	Because <code>TipcSrvCreate</code> calls <code>TipcSrvGmdResend</code> , which will open any existing GMD files and thus not allow them to be deleted, <code>TipcSrvGmdFileDelete</code> cannot create a connection to <code>RTserver</code>

If a user-defined RTclient does not care when it creates a connection to RTserver, it never needs to call `TipcSrvCreate` and can just let the process automatically connect to RTserver when necessary. However, it is usually a good idea to explicitly connect to RTserver once all the necessary options have been set to the desired values.

Automatically Reconnecting to RTserver

When an unrecoverable error occurs on the connection to RTserver, the connection error callbacks for this connection are called (see [Error Callbacks](#), page 110, for more details on connection error callbacks). The standard connection error callback function `TipcCbSrvError` normally takes care of reconnecting to RTserver. `TipcCbSrvError` uses the function `TipcSrvCreate` to reconnect to RTserver so that the same options are used to both connect and reconnect to RTserver. Any messages that were being buffered to be sent to RTserver are lost, but RTclient is able to continue. If the RTclient cannot reconnect to RTserver, it keeps a warm connection to RTserver in the hope that it is able to reconnect to RTserver in the near future. For more information on `TipcCbSrvError` and how it is automatically created as a connection error callback, see the man page for `TipcCbSrvError` in the *TIBCO SmartSockets Application Programming Interface* reference.

While most subject and monitoring information is kept in RTserver, RTclients do keep track of the subjects they are subscribing to and monitoring categories they are watching, so that when they reconnect to RTserver, they can easily start subscribing to those subjects and watching those monitoring categories again.

If several RTclients are connected to RTserver and they all have errors occur on their connections to RTserver (for example, RTserver fails or the node RTserver is running on fails), it is possible that all of the RTclients might try to restart RTserver. These many simultaneous restart attempts can use large amounts of network resources. Avoid this when you configure the option `Server_Names` through the use of the start prefixes (discussed in [Start Prefix](#), page 195), by ensuring that only one or two of the RTclients ever tries to restart RTserver. The default behavior is for RTclients to not attempt to restart the RTserver.

Destroying the Connection to RTserver

If an RTclient is done communicating with RTserver, it can destroy its connection to RTserver. The connection can be fully destroyed, which destroys all RTserver-related information in an RTclient, or it can be partially destroyed, which leaves a subset of the full information in a warm connection. Warm connections are discussed in Warm Connection to RTserver on page 235. From this point on, the term destroying the connection to RTserver is used to mean fully destroying the connection to RTserver.

The function `TipcSrvDestroy` is used to destroy the connection to RTserver. For example:

```
if (!TipcSrvDestroy(T_IPC_SRV_CONN_NONE)) {
    TutOut("Could not destroy connection to RTserver: error <%s>.\n",
          TutErrStrGet());
}
```

The `disconnect` command provides a way to use `TipcSrvDestroy` from the SmartSockets command interface (see the man page for `disconnect` in Chapter 9, Command Reference, for more details). After a process destroys its connection to RTserver it can continue as if it had never been connected to RTserver.

For a full destroy, `TipcSrvDestroy` removes all local subject information, destroys all internal callbacks (for options like `Server_Read_Timeout` and `Log_In_Data`), destroys the connection, and finally calls the server destroy callbacks (see RTclient-Specific Callbacks on page 237 on server destroy callbacks). All messages that have been sent to the connection to RTserver but not flushed are lost after a call to `TipcSrvDestroy`. If a warm connection is retained, all messages that have been read from the connection to RTserver but not processed after the call to `TipcSrvDestroy(T_IPC_SRV_CONN_WARM)` are still available.

`TipcSrvDestroy` also attempts to send a `DISCONNECT` message to notify RTserver of the value of the option `Server_Disconnect_Mode`. `DISCONNECT` messages are discussed in `DISCONNECT` Message Type, page 351.

An RTclient using the TCP protocol to connect to RTserver can lose outgoing messages if the process terminates without calling `TipcSrvDestroy`. TCP/IP's `SO_LINGER` option, which preserves data, is ignored when closing a socket that has non-blocking I/O enabled. While data loss is rare on UNIX, MVS, and OpenVMS, it can happen frequently on Windows. `TipcSrvDestroy` sets the block mode of the connection to `FALSE` before closing the connection's socket, forcing the operating system to deliver all flushed outgoing messages.

Using Subjects

Subscribing to a Subject

The function `TipcSrvSubjectSetSubscribe` can be used to start or stop subscribing to a subject. This code causes RTclient to start subscribing to the subject named `/elec_pwr`:

```
if (!TipcSrvSubjectSetSubscribe("/elec_pwr", TRUE)) {
    TutOut("Could not start subscribing to the /elec_pwr
          subject.\n");
    TutOut("    error <%s>.\n", TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

Once RTclient starts subscribing to a subject, it receives all messages sent to RTserver that have that subject as their destination property. If the second parameter to `TipcSrvSubjectSetSubscribe` is `FALSE`, then RTclient stops subscribing to the subject. `TipcSrvSubjectSetSubscribe` causes a `SUBJECT_SET_SUBSCRIBE` message to be sent to RTserver, but the message is not automatically flushed. See [Sending Messages](#) on page 208 for a discussion of sending and buffering messages to RTserver.

The function `TipcSrvSubjectGetSubscribe` can be used to determine whether or not RTclient is subscribing to a subject. For example:

```
T_BOOL subscribe_status;

if (!TipcSrvSubjectGetSubscribe("/elec_pwr", &subscribe_status)) {
    /* error */
}

TutOut("This process %s subscribing to the /elec_pwr subject.\n",
      subscribe_status ? "is" : "is not");
```

The `subscribe` and `unsubscribe` commands provide a way to use `TipcSrvSubjectSetSubscribe` from the SmartSockets command interface. See [subscribe](#), page 626 and [unsubscribe](#), page 630.

Monitoring a Subject

There are many types of information about subjects that can be monitored. For information on monitoring, see [Chapter 5, Project Monitoring](#).

Unique Subject

As described in Unique Subject, page 162, an RTclient has a unique subject, which RTserver requires to be unique among all processes in a project. This code shows how an RTclient can use this property to ensure that it is the only process subscribing to a subject:

```
TutCommandParseStr("setopt unique_subject my_subject");
if (!TipcSrvCreate(T_IPC_SRV_CONN_FULL)
    && TutErrNumGet() == T_ERR_SRV_ACCESS_DENIED) {
    TutOut("Another RTclient is using my_subject!\n");
}
```

Because the unique subject must be unique among all processes in a project, it can be used to prevent multiple similarly-configured processes from running (whether accidentally or intentionally). Because the default value for the unique subject is generated from the node name and process identifier of the RTclient, the default value is usually adequate for RTclients that aren't using guaranteed message delivery. Configuring GMD, page 331 discusses how Unique_Subject must be explicitly set to use file-based GMD.

RTclient automatically subscribes to its unique subject when it connects to RTserver, and SmartSockets does not allow RTclient to ever stop subscribing to its unique subject.

Standard Subjects

The function TipcSrvStdSubjectSetSubscribe can be used to start or stop subscribing to the standard subjects. This code causes RTclient to start subscribing to all standard subjects, including the `_time` subject:

```
if (!TipcSrvStdSubjectSetSubscribe(TRUE, TRUE)) {
    /* error */
}
```

If the first parameter to TipcSrvStdSubjectSetSubscribe is `FALSE`, then RTclient stops subscribing to the standard subjects. If the second parameter to TipcSrvStdSubjectSetSubscribe is `FALSE`, then the `_time` subject is not included with the standard subjects.

Subject Callbacks

Subject callbacks are functions that are executed while processing a message with the given destination (that is, subject). Subject callbacks are similar to connection process callbacks, but have added flexibility to filter the callbacks based on destination and message type. The functions `TipcSrvSubjectCbCreate` and `TipcSrvSubjectDefaultCbCreate` can be used to create subject callbacks. The following code causes RTclient to create a subject callback that gets executed when any type of message is received by the `"/stocks"` subject:

```
if (!TipcSrvSubjectCbCreate("/stocks", NULL, subject_cb, NULL)) {
    /* error */
}
```

See [RTclient-Specific Callbacks](#) on page 237 for more information on this subject callbacks.

Callbacks

Once RTclient has connected to RTserver, it can create callbacks to be executed when certain operations occur. Most of the connection callback types, except encode and decode callbacks, are available with the connection to RTserver. These connection callback types are discussed in detail in the section [Callbacks](#) on page 84. In addition to the connection callbacks, subject, server create, server destroy, and server names traverse callbacks are available for the connection to RTserver.

For the connection to RTserver, callbacks are manipulated with `TipcSrv*` functions instead of `TipcConn*` functions. The `TipcSrv*CbCreate` functions create callbacks in the connection to RTserver, and the `TipcSrv*CbLookup` functions look up existing callbacks.

This creates a process callback called whenever a `NUMERIC_DATA` message is processed:

```
mt = TipcMtLookupByNum(T_MT_NUMERIC_DATA);
if (mt == NULL) {
    TutOut("Could not look up NUMERIC_DATA msg type: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (TipcSrvProcessCbCreate(mt, cb_process_numeric_data, NULL)
    == NULL) {
    TutOut("Could not create NUMERIC_DATA process cb: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

The callback functions themselves are written exactly the same for the connection to RTserver as for any other connection. The following code shows a callback function that prints the values in a NUMERIC_DATA message. The definition and prototype for every callback function must use the T_ENTRY macro for cross-platform compatibility.

```
/* ===== */
/*..cb_process_numeric_data -- process callback for NUMERIC_DATA */
static void T_ENTRY cb_process_numeric_data(
    T_IPC_CONN conn,
    T_IPC_CONN_PROCESS_CB_DATA data,
    T_CB_ARG arg)
{
    T_STR name;
    T_REAL8 value;

    TutOut("Entering cb_process_numeric_data.\n");

    /* set current field to first field in message */
    if (!TipcMsgSetCurrent(data->msg, 0)) {
        TutOut("Could not set current field of message: error <%s>.\n",
            TutErrStrGet());
        return;
    }
    /* access and print fields */
    while (TipcMsgNextStrReal8(data->msg, &name, &value)) {
        TutOut("%s = %s\n", name, TutRealToStr(value));
    }
    /* make sure we reached the end of the message */
    if (TutErrNumGet() != T_ERR_MSG_EOM) {
        TutOut("Did not reach end of message: error <%s>.\n",
            TutErrStrGet());
    }
} /* cb_process_numeric_data */
```

These callbacks can be created either before or after RTclient starts working with subjects, but the callbacks should be created before any messages are sent or received (if the callbacks are needed for those messages).

RTclient also has callbacks that can be called when certain RTclient-specific events occur, such as when the connection to RTserver is created. See RTclient-Specific Callbacks on page 237 for more information on these callback types.

Receiving and Processing Messages

RTclient receives and processes messages from the connection to RTserver the same way that non-RTserver connections are used. See Receiving and Processing Messages on page 115, for more details. As with callbacks, the TpcSrv* functions are used, not the TpcConn* functions.

For example:

```
/* If an error occurs, then TpcSrvMainLoop will restart RTserver */
/* and return FALSE. We can safely continue. */
for (;;) {
    if (!TpcSrvMainLoop(T_TIMEOUT_FOREVER)) {
        TutOut("TpcSrvMainLoop failed: error <%s>.\n",
            TutErrStrGet());
    }
}
```

The above loop handles the case where RTclient loses its connection to RTserver and reconnects again, and simply prints some diagnostic output each time this occurs. Each program can add more or less error checking as desired.

Processing CONTROL Messages

When RTclient creates a connection to RTserver, the function TpcSrvCreate also creates a connection process callback for CONTROL messages using the standard callback function TpcCbSrvProcessControl. This callback function uses the value of the option Enable_Control_Msgs to check if the command is enabled, and if allowed, RTclient then calls TutCommandParseStr to have the process command interface execute the command. This automatic processing of CONTROL messages allows RTclient to publish remote commands to any other RTclient. See Enable_Control_Msgs, page 533 for details on how to properly configure CONTROL message security.

For more information on TpcCbSrvProcessControl and how it is automatically created as a connection process callback, see the reference page for TpcCbSrvProcessControl in the *TIBCO SmartSockets Application Programming Interface* reference.

Sending Messages

RTclient sends (publishes) messages using the connection to RTserver in a way that is similar, but not quite identical, to the way other connections are used. The functions `TipcSrvMsgSend`, `TipcSrvMsgWrite`, and `TipcSrvMsgWriteVa` differ from their `TipcConn*` relatives in these ways:

- The above `TipcSrv*` functions all set the sender property of the message being sent to the value in the option `Unique_Subject`.
- The above `TipcSrv*` functions have an additional *check_server_msg_send* parameter (of type `T_BOOL`) that determines whether or not the option `Server_Msg_Send` is checked before actually sending the message.
- `TipcSrvMsgWrite` and `TipcSrvMsgWriteVa` have an additional *destination* parameter (of type `T_STR`) that is the name of a subject to be used as the destination property of the message being sent (to use `TipcSrvMsgSend` the destination property of the message must be set first with `TipcMsgSetDest`).

The `Server_Msg_Send` option specifies whether or not RTclient should send messages to RTserver. If *check_server_msg_send* is `TRUE` and `Server_Msg_Send` is `FALSE`, then the above `TipcSrv*` functions do not send the message. Some messages sent internally by the SmartSockets IPC library, such as `SUBJECT_SET_SUBSCRIBE` messages, are always sent regardless of the setting of `Server_Msg_Send`. This option is useful for backup processes that should receive messages from RTserver but not send any out. The use of any option greatly simplifies the development of such backup processes; for a complete example see *Running an RTclient With a Hot Backup*, page 429. You should normally use `TRUE` for the *check_server_msg_send* parameter to the above `TipcSrv*` functions if you wish to use the `Server_Msg_Send` option to easily create backup processes.

This uses `TipcSrvMsgWrite` to send an INFO message to all processes in a project:

```
TutOut("Publishing a CONTROL message to stop the receiver(s).\n");
mt = TipcMtLookupByNum(T_MT_CONTROL);
if (mt == NULL) {
    TutOut("Could not look up CONTROL msg type: error <%s>.\n",
          TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcSrvMsgWrite("_ie", mt, TRUE,
                    T_IPC_FT_STR, "quit force",
                    NULL)) {
    TutOut("Could not publish CONTROL message: error <%s>.\n",
          TutErrStrGet());
}
```

The previous example could be rewritten as follows to use `TipcSrvMsgSend` instead of `TipcSrvMsgWrite`:

```
TutOut("Publishing a CONTROL message to stop the receiver(s).\n");
mt = TipcMtLookupByNum(T_MT_CONTROL);
if (mt == NULL) {
    TutOut("Could not look up CONTROL msg type: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
msg = TipcMsgCreate(mt);
if (msg == NULL) {
    TutOut("Could not create CONTROL message: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TipcMsgSetDest(msg, "_ie")
    || !TipcMsgAppendStr(msg, "quit force")
    || !TipcSrvMsgSend(msg, TRUE)) {
    TutOut("Could not construct and publish CONTROL message.\n");
    TutOut("  error <%s>.\n", TutErrStrGet());
}
```

Buffering of Outgoing Messages

The connection to RTServer has the same auto flush size property that all other connections have for controlling how many bytes of outgoing data are buffered before being automatically flushed. This property can be set with the function `TipcSrvSetAutoFlushSize`, but it can also be accessed with the `Server_Auto_Flush_Size` option. This option can be set in startup command files for greater convenience in all RTclients.

Message File Logging

A message file is a file, in either text format or binary format, containing one or more messages. Through the use of options, RTclient can easily log (that is, write into text message files) the messages that have been sent to or received from RTserver. With this message file logging, the message types are grouped into three standard categories. Any message type can also be added to or removed from these categories with the API functions `TipcSrvLogAddMt` and `TipcSrvLogRemoveMt` (see [Changing Logging Categories](#) on page 214). This section describes the features of message file logging.

Message File Logging Categories

Message types are divided into three categories for the purpose of being logged into message file:

- data Not used by SmartSockets. Available for user-defined messages.
- status Not used by SmartSockets. Available for user-defined messages.
- internal SmartSockets internal messages

All standard message types are described in detail in the section Standard Message Types on page 40.

The standard message types in each message file logging category are:

Data	Internal
BOOLEAN_DATA	CONN_INIT
CONTROL	CONNECT_CALL
ENUM_DATA	CONNECT_RESULT
HISTORY_BOOLEAN_DATA	DISCONNECT
HISTORY_ENUM_DATA	GMD_ACK
HISTORY_NUMERIC_DATA	GMD_DELETE
HISTORY_STRING_DATA	GMD_FAILURE
NUMERIC_DATA	GMD_INIT_CALL
OBJECT_DATA	GMD_INIT_RESULT
STRING_DATA	GMD_NACK
VAR_VALUE_CALL	GMD_STATUS_CALL
VAR_VALUE_RESULT	GMD_STATUS_RESULT
	KEEP_ALIVE_CALL
	KEEP_ALIVE_RESULT
	MON_* (64 msg types)
	SERVER_STOP_CALLSERVER_STOP_RESULT
	SUBJECT_RETRIEVE
	SUBJECT_SET_SUBSCRIBE

Data Messages

Data messages are the most common category of message sent between RTclients. This message file shows several sample data messages (note that alias names are identifiers):

```
time /_time 42
numeric_data /system/thermal {
    voltage 33.4534
    switch_pos 0
}
boolean_data /elec_pwr relay_aligned true
string_data /system {
    i80_message "Carry tire chains"
    i50_message "Road closed"
}
enum_data /system/console battery_status normal
control /system "setopt frame_interval 1.0"
end_of_frame /_time
```

Internal Messages

All the remaining standard message types are included in the internal category. Internal messages include (but are not limited to) these groups of message types:

- messages used to implement GMD
- monitoring messages
- messages sent to and from RTserver to check the health of the connection
- messages sent from RTclient to RTserver to manipulate subjects

You do not usually explicitly create and send internal messages, but logging these types of messages can be useful for debugging purposes. This message file shows several sample internal messages:

```
subject_set_subscribe /_server thermal true true
server_stop_call /_server 2
gmd_ack /_server 345613
server_stop_result /_client "RTserver stopping"
mon_subject_subscribe_status /_client "/_workstation1_2269" {
    "/_workstation1_2269" } "" ""
```

Logging Messages

RTclient starts and stops logging messages in the logging categories by setting these options:

To Log this Category:	Use the Option:
Incoming data messages	Log_In_Data
Outgoing data messages	Log_Out_Data
Incoming status messages	Log_In_Status
Outgoing status messages	Log_Out_Status
Incoming internal messages	Log_In_Internal
Outgoing internal messages	Log_Out_Internal

RTserver can also log incoming and outgoing messages to message files. See Message File Logging on page 295 for more information on logging messages in an RTserver.

Starting Message Logging

Message logging is started by setting one of the above options (either with the function TutOptionSetStr or the `setopt` command) to the name of a file. For example:

```
setopt log_in_data incoming.msg
```

The above example causes the RTclient to start logging all incoming data messages into the `incoming.msg` file. If the file does not exist, it is created. If the file exists from a previous session, it is overwritten with new information.

Multiple categories can be logged into the same file. For example, RTclient could log both incoming data messages and outgoing status messages into the same message file by setting these options:

```
setopt log_in_data messages.msg
setopt log_out_status messages.msg
```

The first line either creates the file or overwrites an existing file and begins logging incoming data messages. The second line begins logging outgoing status messages to the same file.

Stopping Message Logging

Message logging is stopped by setting one of the above options to UNKNOWN, either with the function `TutOptionSetUnknown` or the `unsetopt` command. For example:

```
unsetopt log_in_data
```

The above example causes the `RTclient` to stop logging all incoming data messages. If multiple categories are being logged to the same file, the file is not closed until logging is stopped for all the relevant categories.

Changing Logging Categories

Any message type can be added to or removed from a message file logging category, including both standard message types and user-defined message types. All the standard messages types are automatically added to exactly one category. Message types can be added to multiple categories and they can also be removed from all categories.

The function `TipcSrvLogAddMt` is used to add a message type to a logging category. This creates a user-defined message type and adds it to the data category:

```
#define XYZ_COORD_DATA 1001

mt = TipcMtCreate("xyz_coord_data, XYZ_COORD_DATA, "int4 int4
int4");
if (mt == NULL) {
    /* error */
}
if (!TipcSrvLogAddMt(T_IPC_SRV_LOG_DATA, mt)) {
    /* error */
}
```

The function `TipcSrvLogRemoveMt` is used to remove a message type from a logging category. This looks up the `TIME` message type and removes it from the data category:

```
mt = TipcMtLookupByNum(T_MT_TIME);
if (mt == NULL) {
    /* error */
}
if (!TipcSrvLogRemoveMt(T_IPC_SRV_LOG_DATA, mt)) {
    /* error */
}
```

Both `TipcSrvLogAddMt` and `TipcSrvLogRemoveMt` can only be used before `RTclient` has created any connection to `RTserver`.

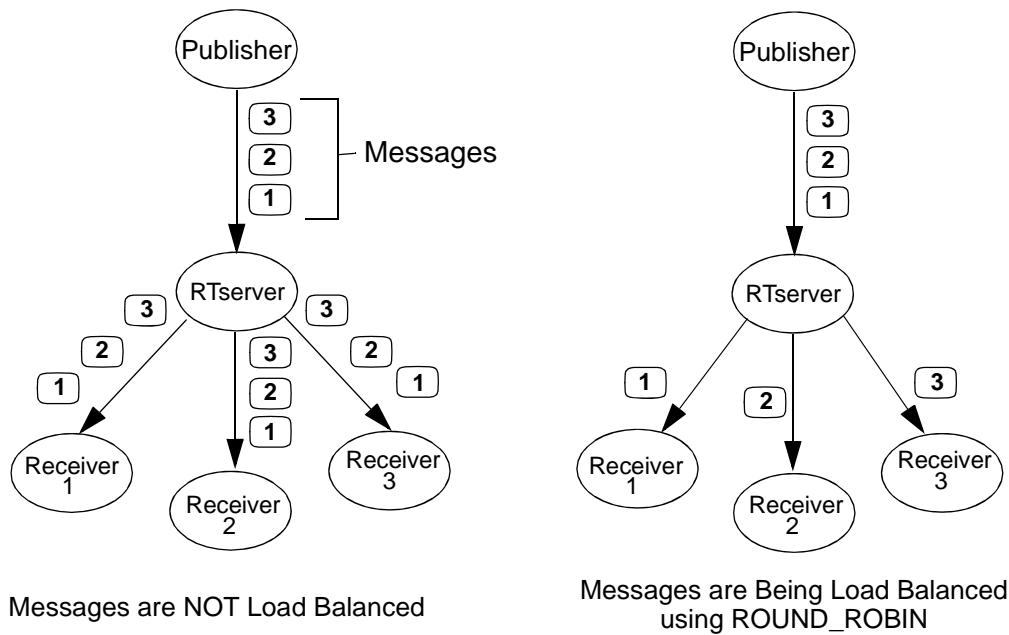
Load Balancing

In normal publish-subscribe operations, a message is sent to all RTclients that have subscribed to the subject the message is being published to. However, in some situations you may wish to have messages sent to only one subscribing RTclient. An example of this is a project where there is high message throughput and each message takes some time to process. In this case, you may wish to replicate a set of RTclients and have them take turns processing the messages to better keep up with message flow.

This is accomplished in SmartSockets using load balancing. Rather than have a single RTclient handle all the messages, you can use load balancing to process the messages across multiple RTclients. This is very useful when processing a heavy message load. A load-balanced message is routed to only a single RTclient, not to all RTclients subscribed to the destination subject. The RTclient to which the message is routed is selected based on the load balancing mode specified. Load balancing implies that there is a set of RTclients that are all equally capable of processing load-balanced messages.

For example, consider the simple example shown in Figure 18. There are three receivers, all subscribed to the same subject. Messages 1, 2, and 3 are published to that subject. On the left side of the figure, each message is routed to all receivers because there is no load balancing. The right side shows what happens when the messages are marked to be delivered using round-robin load balancing. The first message is delivered to Receiver 1, the second message to Receiver 2, and the third message to Receiver 3. Each message is delivered to only a single RTclient.

Figure 18 Messages Delivered With and Without Load Balancing



Load balancing can be specified per-message or per-message type using the load balancing mode message property (see Load Balancing Mode on page 15 for more information). Setting the load balancing mode for each message takes precedence over per-message type. By default, messages are not load balanced and are distributed to all subscribers.

Load balancing is dynamic in that whenever an RTclient connects or disconnects to an RTserver, the load balancing calculations are updated in real time. When an RTclient publishes the first message using load balancing to a subject, RTserver starts collecting subject subscription information from the appropriate RTservers to accurately track load balancing accounting. This increases the scalability of load balancing due to the fact that only the relevant RTservers dynamically exchange load balancing information. The RTclient API function `TipcSrvSubjectGmdInit` can also be used to manually initialize GMD accounting for a subject to which messages will be published.

Overriding Load Balancing

It is important to note that any RTclient has the ability to override load balancing, on a subject basis, so an RTclient could subscribe to a subject that has load-balanced messages being sent to it, but receive all the messages, such as for an archive). A simple example would be if you are monitoring a project and wish to see all the messages sent to a specific subject. If you did a normal subscribe and the messages were being load balanced, you would not see each one if there was another RTclient subscribed to the same subject in the project. To override load balancing, the function `TipcSrvSubjectSetSubscribeLb(subject_name, TRUE, FALSE)` should be used instead of `TipcSrvSubjectSetSubscribe(subject_name, TRUE)`. The first parameter specifies to which subject you are subscribing or unsubscribing. The second parameter specifies whether you are subscribing (TRUE) or unsubscribing (FALSE). The last parameter specifies whether you wish to receive all messages (FALSE) or be included in the load balancing calculations (TRUE). For example, an RTclient wishing to receive all messages published to the subject `"/manual/chapter4"`, regardless of whether the messages are load balanced or not, would call the function as follows:

```
TipcSrvSubjectSetSubscribeLb("/manual/chapter4", TRUE, FALSE);
```

The subscribe command can also be used with the `-load_balancing_off` parameter to override load balancing:

```
MON> subscribe -load_balancing_off "/manual/chapter4"
```

Load Balancing Modes

SmartSockets supports several load balancing modes as shown in Table 10.

Table 10 Load Balancing Modes

Mode	Description
T_IPC_LB_NONE	<p>This is the default and specifies no load balancing. The message is sent to all subscribers.</p>
T_IPC_LB_ROUND_ROBIN	<p>The list of subscribing RTclients is held in a circular list, with each successive message simply sent to the next RTclient in the list.</p> <p>This mode is a good choice when the subscribers are all capable of receiving and processing a request with nearly equal speed. There is no additional overhead with this mode.</p>
T_IPC_LB_WEIGHTED	<p>The message is published to the RTclient that has the fewest pending requests.</p> <p>This mode is a good choice when the subscribers differ significantly in their ability to process a request promptly, such as hardware speed differences or network delays.</p> <p>This method can only be used with GMD and requires no additional overhead beyond what GMD requires.</p>
T_IPC_LB_SORTED	<p>The message is always sent to the first RTclient in the list. The list is formed by doing an alphabetical sort of the unique subject name of each RTclient.</p> <p>This mode is a good choice when you want a specific subscriber to process all messages until it fails, when a hot standby can take over. There is no additional overhead with this mode.</p>

Weighted load balancing takes into account the receiver’s ability to process messages using acknowledgments. These acknowledgments are a result of a message being sent guaranteed. Weighted load balancing can only be used with GMD. When RTclient publishes a message that uses weighted load balancing, RTserver cycles through the list of receivers and sends the message to the first RTclient which has the fewest number of unacknowledged messages. Using the number of unacknowledged messages takes into account the speed of the receiver and the speed of the round trip to that receiver.

For example, with a message stream that requires significant CPU processing resources, a fast receiver only one RTserver hop away would be able to process and acknowledge messages faster then a slow receiver several RTserver hops away. As a result, the fast receiver which is closet would often be favored over the

slower receiver which would not be able to acknowledge messages as quickly. So, the weighting of receivers by the number of acknowledgments takes into account not just the speed of the receiver's system, but also the speed of systems running RTservers in between the publisher and subscriber, the networks used to connect the systems, and any overhead that could be introduced by the load balancing algorithm.

See Multiple RTserver Processes on page 298 for more details on using multiple RTservers on the same project.

Load Balancing and GMD

A message that is being load balanced can also be guaranteed (have a delivery mode of `SOME` or `ALL`; see Chapter 4, Guaranteed Message Delivery for more information). If a message is to be both load balanced and guaranteed, message delivery failures are handled in the following ways.

- If the connection to an RTclient is lost, the RTclient's `Server_Disconnect_Mode` option is set to `warm`, and it reconnects before the amount of time specified in the `Client_Reconnect_Timeout` expires, then the message is resent by RTserver to the RTclient when it reconnects.
- If the connection to an RTclient is lost, the RTclient's `Server_Disconnect_Mode` option is set to `warm`, and it does not reconnect before the amount of time specified in the `Client_Reconnect_Timeout` expires, then the message is load balanced among the remaining RTclients if it is resent by the publisher.

Note that `ROUND_ROBIN` and `SORTED` can be used with or without GMD. `WEIGHTED` requires GMD to be used as RTserver uses the GMD acknowledgments in its load balancing calculations. A warm RTclient is not counted in the load balancing calculation unless all subscribing RTclients are warm RTclients (see Warm RTclient in RTserver, page 346).

Load Balancing Example

This example shows a publishing RTclient that creates and sends 20 messages. The first 10 messages are sent with a load balancing mode of `NONE`, so all subscribers receive them. The next 10 messages are sent with a load balancing mode of `ROUND_ROBIN` so they are evenly distributed across the subscribers.

The source code files for this example are located in these directories:

UNIX:

```
$RTHOME/examples/smrtsock/manual
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]
```

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

The online source files have additional `#ifdefs` to provide C++ support; these `#ifdefs` are not shown to simplify the example.

The subscribers simply print out the contents of the messages. The source code for the subscribers is not shown, but the `lbrecv.c` file is located in the same directory as the sender program.

Example 19 Sender Source Code

```
/* lbsend.c - send messages (some load balanced, some not) */
```

```
#include <rtworks/ipc.h>
#define MSG_COUNT 1001

int main(argc, argv)
int argc;
char **argv;
{
    T_OPTION option;
    T_IPC_MSG msg;
    T_IPC_MT mt;
    T_INT4 i;
```

```

/* Set the option Project to partition ourself. */
option = TutOptionLookup("project");
if (option == NULL) {
    TutOut("Could not look up option named project: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TutOptionSetEnum(option, "smartsockets")) {
    TutOut("Could not set option named project: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* Define a new message type */
mt = TipcMtCreate("msg_count", MSG_COUNT, "int4");
if (mt == NULL) {
    TutOut("Could not create message type MSG_COUNT: error
<%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* Connect to RTserver */
if (!TipcSrvCreate(T_IPC_SRV_CONN_FULL)) {
    TutOut("Could not connect to RTserver: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* Create a message of type MSG_COUNT */
msg = TipcMsgCreate(mt);
if (msg == NULL) {
    TutOut("Could not create msg of type MSG_COUNT: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* Set the destination subject of the message */
if (!TipcMsgSetDest(msg, "/manual/chapter4")) {
    TutOut("Could not set subject of message: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

for (i = 0; i < 20; i++) {
/* Reset num of fields to 0 so we can reuse message */
    if (!TipcMsgSetNumFields(msg, 0)) {
        TutOut("<%d> Could not clear message: error <%s>.\n",
            i, TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
}

```

```

/*
 * First 10 messages, send to everyone,
 * Second 10 messages, load balance using round robin
 */
    if (i < 10) {
        /* This is the default behavior and not required */
        if (!TipcMsgSetLbMode(msg, T_IPC_LB_NONE)) {
            TutOut("Could not set load balance to NONE: error <%s>.\n",
                TutErrStrGet());
        }
    }
    else {
        if (!TipcMsgSetLbMode(msg, T_IPC_LB_ROUND_ROBIN)) {
            TutOut("Could not set load balance to ROUND_ROBIN\n");
            TutOut("  error <%s>.\n", TutErrStrGet());
        }
    }

    /* Build the data part of the message with 1 integer */
    if (!TipcMsgAppendInt4(msg, i)) {
        TutOut("<%d> Could not build message: error <%s>.\n",
            i, TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    /* Publish the message */
    if (!TipcSrvMsgSend(msg, TRUE)) {
        TutOut("<%d> Could not publish message: error <%s>.\n",
            i, TutErrStrGet());
    }

    /* Make sure message is flushed */
    if (!TipcSrvFlush()) {
        TutOut("<%d> Could not flush message: error <%s>.\n",
            i, TutErrStrGet());
    }

}

/* Destroy the message */
if (!TipcMsgDestroy(msg)) {
    TutOut("<%d> Could not destroy message: error <%s>.\n",
        i, TutErrStrGet());
}

return T_EXIT_SUCCESS; /* all done */
} /* main */

```


Compiling, Linking, and Running

To compile, link, and run the example programs, first you must either copy the programs to your own directory or have write permission in these directories:

UNIX:

```
$RTHOME/examples/smrtsock/manual
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]
```

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

Step 1 To compile and link the program, use:

UNIX:

```
$ rtlink -o lbsend.x lbsend.c
$ rtlink -o lbrecv.x lbrecv.c
```

OpenVMS:

```
$ cc lbsend.c, lbrecv.c
$ rtlink /exec=lbSEND.exe lbsend.obj
$ rtlink /exec=lbRCV.exe lbrecv.obj
```

Windows:

```
$ nmake /f lbsdw32m.mak
$ nmake /f lbrcw32m.mak
```

Step 2 Start an RTserver

To see how the load balancing works, start your RTserver.

Then you start two copies of `lbrecv` and start the publishing process, `lbSEND`. To do this, execute the start command in two separate windows.

Step 3 Start lbrecv

UNIX:

```
$ lbrecv.x
```

OpenVMS:

```
$ run lbrecv.exe
```

Windows:

```
$ lbrecv.exe
```

Step 4 Start lbsend

To run the sending program, use:

UNIX:

```
$ lbsend.x
```

OpenVMS:

```
$ run lbsend.exe
```

Windows:

```
$ lbsend.exe
```

An example of the output from the window where the first lbrecv is running is shown:

```
Connecting to project <smartsockets> on <_node> RTserver
Using local protocol
Message from RTserver: Connection established.
Start subscribing to subject </_workstation.talarian.com_24000>
Message data = 0
Message data = 1
Message data = 2
Message data = 3
Message data = 4
Message data = 5
Message data = 6
Message data = 7
Message data = 8
Message data = 9
Message data = 11
Message data = 13
Message data = 15
Message data = 17
Message data = 19
```

An example of the output from the window where the second `lbrecv` is running is shown:

```
Connecting to project <smartsockets> on <_node> RTserver.
Using local protocol.
Message from RTserver: Connection established.
Start subscribing to subject </_workstation.talarian.com_24003>.
Message data = 0
Message data = 1
Message data = 2
Message data = 3
Message data = 4
Message data = 5
Message data = 6
Message data = 7
Message data = 8
Message data = 9
Message data = 10
Message data = 12
Message data = 14
Message data = 16
Message data = 18
```

Note that both programs processed the first ten messages. This is because the load balancing mode of these messages was set to `NONE`. The next ten messages were evenly distributed across the two processes because load balancing mode was set to `ROUND_ROBIN` for these messages. The first `lbrecv` processed messages 11, 13, 15, 17, and 19 and the second `lbrecv` processed messages 10, 12, 14, 16, and 18.

As an interesting exercise to see how an `RTclient` can still receive all messages, whether they are being load balanced or not, make a copy of the `lbrecv` program. Change the line in the main program from:

```
if (!TipcSrvSubjectSetSubscribe("/manual/chapter4", TRUE)) {
to:
if (!TipcSrvSubjectSetSubscribeLb("/manual/chapter4", TRUE,
FALSE)) {
```

Compile, link, and run the new program alongside two copies of `lbrecv`. You see that the new program processes all 20 messages, while the two copies of `lbrecv` behave just as they did before.

Using Threads with the RTclient API

As described in Using Threads With Connections on page 125, multithreading is an effective technique for SmartSockets applications. RTclients can be multithreaded as well, using a warm or full connection to RTserver for high-level synchronization.

In addition to the synchronization properties incorporated into each connection, the RTserver connection is protected by an additional read/write mutex. This extra level of synchronization enables one thread in an RTclient to change the state of the RTserver connection without interfering with the other threads sharing the connection. Operations that do not affect the state of the RTserver connection are performed with only a read-lock on the read/write mutex and may thus run concurrently, while operations that do affect the state acquire a write-lock.

Working with threads in an RTclient is very similar to working with threads in peer-to-peer connections. The source code files, `rtsvcp.c` and `rtsvcc.c`, illustrating load balancing and threads are located in these directories:

UNIX:

```
$RTHOME/examples/smrtsock/manual
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]
```

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

Advanced RTclient Usage

This section covers advanced usage of RTclients, and many topics might cover issues that are not relevant to your applications. Advanced usage adds another layer of complication, so be sure you need the feature before designing your application around it. In other words, the simplest RTclient that does the job is the best for the application. It will be easier to debug and to maintain.

Advanced Example With Warm Connections and Server Callbacks

This example shows several advanced features of RTclient, including a warm connection to RTserver, server create callbacks, and server destroy callbacks. To learn more about working with the basic features of RTclient, see *Working With RTclient* on page 175.

The source code files for this example are located in these directories:

UNIX:

```
$RTHOME/examples/smrtsock/manual
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]
```

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

The online source files have additional `#ifdefs` to provide C++ support; these `#ifdefs` are not shown to simplify the example.

Advanced Example Source Code

Here is an example of the advanced RTclient code:

```
/* rtcladv.c -- RTclient advanced example */

/*
   This RTclient creates and destroys its connection to RTserver in various ways, and publishes
   messages to a subject it is subscribing to.
*/

#include <rtworks/ipc.h>
#include "rtclient.h"
```

```

/* ===== */
/*..cb_default -- default callback */
static void T_ENTRY cb_default(
    T_IPC_CONN conn,
    T_IPC_CONN_DEFAULT_CB_DATA data,
    T_CB_ARG arg)
{
    T_IPC_MT mt;
    T_STR name;

    TutOut("Entering cb_default.\n");

    /* print out the name of the type of the message */
    if (!TipcMsgGetType(data->msg, &mt)) {
        TutOut("Could not get message type from message: error
<%s>.\n",
            TutErrStrGet());
        return;
    }
    if (!TipcMtGetName(mt, &name)) {
        TutOut("Could not get name from message type: error <%s>.\n",
            TutErrStrGet());
        return;
    }
    TutOut("Message type name is <%s>\n", name);
} /* cb_default */

/* ===== */
/*..cb_server_create -- server create callback */
static void T_ENTRY cb_server_create(
    T_IPC_CONN conn,
    T_IPC_SRV_CREATE_CB_DATA data,
    T_CB_ARG arg)
{
    /* Create other callbacks if we did not have a warm connection. */
    if (data->old_conn_status == T_IPC_SRV_CONN_WARM) {
        TutOut("We already had a warm connection to RTserver, which ");
        TutOut("preserves all callbacks.\n");
        return; /* nothing to do */
    }

    /* A larger process would create many callbacks here. This */
    /* simple example only creates a connection default callback. */
    TutOut("Creating other callbacks.\n");

    /* default callback */
    if (TipcSrvDefaultCbCreate(cb_default, NULL) == NULL) {
        TutOut("Could not create default callback: error <%s>.\n",
            TutErrStrGet());
    }
} /* cb_server_create */

```

```

/* ===== */
/*..cb_server_destroy -- server destroy callback */
static void T_ENTRY cb_server_destroy(
    T_IPC_CONN conn,
    T_IPC_SRV_DESTROY_CB_DATA data,
    T_CB_ARG arg)
{
    TutOut("Entering cb_server_destroy.\n");
    if (data->new_conn_status == T_IPC_SRV_CONN_WARM) {
        TutOut("Leaving a warm connection to RTserver.\n");
    }
    else {
        TutOut("Leaving no connection to RTserver.\n");
    }
}

/* cb_server_destroy */

/* ===== */
/*..main -- main program */
int main(argc, argv)
int argc;
char **argv;
{
    T_OPTION option;
    T_IPC_MT mt;

    /* Set the option Project to partition ourself. */
    option = TutOptionLookup("project");
    if (option == NULL) {
        TutOut("Could not look up option named project: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
    if (!TutOptionSetEnum(option, EXAMPLE_PROJECT)) {
        TutOut("Could not set option named project: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    /* Allow a command-line argument containing the name of a */
    /* SmartSockets startup command file. This file can be used */
    /* to set options like Server_Names. */
    if (argc == 2) {
        if (!TutCommandParseFile(argv[1])) {
            TutOut("Could not parse startup command file %s: error
<%s>.\n",
                argv[1], TutErrStrGet());
            TutExit(T_EXIT_FAILURE);
        }
    }
    else if (argc != 1) { /* too many command-line arguments */
        TutOut("Usage: %s [ command_file_name ]\n", argv[0]);
        TutExit(T_EXIT_FAILURE);
    }
}

```

```

    /* create callbacks to be executed when certain operations occur */
    TutOut("Create callbacks.\n");

    /* If an RTclient will be creating and destroying its connection */
    /* to RTserver over and over, it should use a server create */
    /* callback to create the rest of its callbacks. */

    /* server create callback */
    if (TipcSrvCreateCbCreate(cb_server_create, NULL) == NULL) {
        TutOut("Could not create server create callback: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    /* server destroy callback */
    if (TipcSrvDestroyCbCreate(cb_server_destroy, NULL) == NULL) {
        TutOut("Could not create server destroy callback: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    TutOut("Creating connection to RTserver.\n");
    if (!TipcSrvCreate(T_IPC_SRV_CONN_FULL)) {
        TutOut("Could not create connection to RTserver: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    /* Destroy connection to RTserver to show how this can be done. */
    TutOut("\nDestroying connection to RTserver but leave it
warm.\n");
    if (!TipcSrvDestroy(T_IPC_SRV_CONN_WARM)) {
        TutOut("Could not destroy connection to RTserver: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    TutOut("Creating connection to RTserver from warm
connection.\n");
    if (!TipcSrvCreate(T_IPC_SRV_CONN_FULL)) {
        TutOut("Could not create connection to RTserver: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
    TutOut("Completely destroying connection to RTserver.\n");
    if (!TipcSrvDestroy(T_IPC_SRV_CONN_NONE)) {
        TutOut("Could not destroy connection to RTserver: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

```



```

/* At this point, the RTclient could proceed as if it had */
/* never been connected to RTserver at all. For this example */
/* we want to reconnect eventually, though. */

/* Set the option Server_Auto_Connect to FALSE to show how */
/* an RTclient can buffer outgoing messages even when not */
/* connected to RTserver. */
    TutCommandParseStr("setopt server_auto_connect false");

    TutOut("\nCreating warm connection to RTserver.\n");
    if (!TipcSrvCreate(T_IPC_SRV_CONN_WARM)) {
        TutOut("Could not create warm connection to RTserver.\n");
        TutOut(" error <%s>.\n", TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    /* Now proceed with normal processing even though we only */
    /* have a warm connection. */
    TutOut("Start subscribing to standard subjects.\n");
    if (!TipcSrvStdSubjectSetSubscribe(TRUE, TRUE)) {
        TutOut("Could not start subscribing to standard subjects.\n");
        TutOut(" error <%s>.\n", TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    TutOut("Start subscribing to the %s subject.\n",
EXAMPLE_SUBJECT);
    if (!TipcSrvSubjectSetSubscribe(EXAMPLE_SUBJECT, TRUE)) {
        TutOut("Could not start subscribing to %s subject: error
<%s>.\n",
        EXAMPLE_SUBJECT, TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    TutOut("Publish a frame of data to the receiver's subject.\n");

    TutOut("Publishing a TIME message.\n");
    mt = TipcMtLookupByNum(T_MT_TIME);
    if (mt == NULL) {
        TutOut("Could not look up TIME msg type: error <%s>.\n",
        TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
    if (!TipcSrvMsgWrite(EXAMPLE_SUBJECT, mt, TRUE,
        T_IPC_FT_REAL8, 1.0,
        NULL)) {
        TutOut("Could not publish TIME message: error <%s>.\n",
        TutErrStrGet());
    }
}

```

```

    TutOut("Publishing a NUMERIC_DATA message.\n");
    mt = TipcMtLookupByNum(T_MT_NUMERIC_DATA);
    if (mt == NULL) {
        TutOut("Could not look up NUMERIC_DATA msg type: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
    if (!TipcSrvMsgWrite(EXAMPLE_SUBJECT, mt, TRUE,
        T_IPC_FT_STR, "voltage",
        T_IPC_FT_REAL8, 33.4534,
        T_IPC_FT_STR, "switch_pos",
        T_IPC_FT_REAL8, 0.0,
        NULL)) {
        TutOut("Could not publish NUMERIC_DATA message: error <%s>.\n",
            TutErrStrGet());
    }

    TutOut("Publishing an END_OF_FRAME message.\n");
    mt = TipcMtLookupByNum(T_MT_END_OF_FRAME);
    if (mt == NULL) {
        TutOut("Could not look up END_OF_FRAME msg type: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
    if (!TipcSrvMsgWrite(EXAMPLE_SUBJECT, mt, TRUE, NULL)) {
        TutOut("Could not publish END_OF_FRAME message: error <%s>.\n",
            TutErrStrGet());
    }

    TutOut("\nCreating connection to RTserver.\n");
    if (!TipcSrvCreate(T_IPC_SRV_CONN_FULL)) {
        TutOut("Could not create connection to RTserver: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    /* Each RTclient automatically creates a connection process */
    /* callback for CONTROL messages. Use this to send the command */
    /* "quit force" to the receiver's command interface. */
    TutOut("Publishing a CONTROL message to stop ourself.\n");
    mt = TipcMtLookupByNum(T_MT_CONTROL);
    if (mt == NULL) {
        TutOut("Could not look up CONTROL msg type: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
    if (!TipcSrvMsgWrite(EXAMPLE_SUBJECT, mt, TRUE,
        T_IPC_FT_STR, "quit force",
        NULL)) {
        TutOut("Could not publish CONTROL message: error <%s>.\n",
            TutErrStrGet());
    }
}

```

```

/* Flush the buffered outgoing messages to RTserver. */
if (!TipcSrvFlush()) {
    TutOut("Could not flush messages to RTserver: error <%s>.\n",
        TutErrStrGet());
}

/* If an error occurs, then TipcSrvMainLoop will restart RTserver */
/* and return FALSE. We can safely continue. */
for (;;) {
    if (!TipcSrvMainLoop(T_TIMEOUT_FOREVER)) {
        TutOut("TipcSrvMainLoop failed: error <%s>.\n",
            TutErrStrGet());
    }
}

/* This line should not be reached. */
TutOut("This line should not be reached!!!\n");
return T_EXIT_FAILURE;
} /* main */

```

Compiling, Linking, and Running

To compile, link, and run the example programs, first you must either copy the programs to your own directory or have write permission in these directories:

UNIX:

```
$RTHOME/examples/smrtsock/manual
```

OpenVMS

```
RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]
```

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

Step 1 **To compile and link the program, use:**

UNIX:

```
$ rtlink -o rtcladv.x rtcladv.c
```

OpenVMS:

```
$ cc rtcladv.c
$ rtlink /exec=rtcladv.exe rtcladv.obj
```

Windows:

```
$ nmake /f cladw32m.mak
```

Step 2 **Ensure RTserver is running**

Step 3 To run the program, use:**UNIX:**

```
$ rtcladv.x
```

OpenVMS:

```
$ run rtcladv.exe
```

Windows:

```
$ rtcladv.exe
```

Here is an example of the output:

```
Create callbacks.
Creating connection to RTserver.
Connecting to project <example> on <_node> RTserver
Using local protocol
Message from RTserver: Connection established.
Start subscribing to subject </_workstation_5415>
Creating other callbacks.

Destroying connection to RTserver but leave it warm.
Entering cb_server_destroy.
Leaving a warm connection to RTserver.
Creating connection to RTserver from warm connection.
Attempting to reconnect to RTserver
Connecting to project <example> on <_node> RTserver
Using local protocol
Message from RTserver: Connection established.
Start subscribing to subject </_workstation_5415> again
We already had a warm connection to RTserver, which preserves all
callbacks.
Completely destroying connection to RTserver.
Entering cb_server_destroy.
Leaving no connection to RTserver.

Creating warm connection to RTserver.
Creating other callbacks.
Start subscribing to standard subjects.
Start subscribing to subject </_time>
Start subscribing to subject </_workstation_5415>
Start subscribing to subject </_all>
Start subscribing to the rcv subject.
Publish a frame of data to the receiver's subject.
Publishing a TIME message.
Publishing a NUMERIC_DATA message.
Publishing an END_OF_FRAME message.
```

```

Creating connection to RTserver.
Attempting to reconnect to RTserver.
Connecting to project <example> on <_node> RTserver
Using local protocol
Message from RTserver: Connection established.
Start subscribing to subject </_workstation_5415> again.
Start subscribing to subject </_all> again.
Start subscribing to subject </_workstation> again.
Start subscribing to subject </_time> again.
Start subscribing to subject </_rcv> again.
We already had a warm connection to RTserver, which preserves all
callbacks.
Publishing a CONTROL message to stop ourself.
Entering cb_default.
Message type name is <time>
Entering cb_default.
Message type name is <numeric_data>
Entering cb_default.
Message type name is <end_of_frame>

```

Warm Connection to RTserver

A warm connection to RTserver is a subset of a full connection to RTserver. A warm connection keeps as much RTserver-related information as possible. The only difference between a warm connection and a full connection is that the warm connection does not have a valid socket (there is no communication link to RTserver with a warm connection). No messages can be flushed to RTserver on a warm connection and no messages can be read from the warm connection, but most TipcSrv* functions behave in a fashion similar to when a full connection exists. RTserver is not aware of the RTclient when the RTclient has a warm connection, unless RTclient has told RTserver to keep a warm RTclient record for this RTclient (see Warm RTclient in RTserver, page 346).

With a warm connection to RTserver, callbacks can be created, callbacks can be destroyed, and messages can be buffered. If RTclient has a warm connection and then creates a full connection (the connection conceptually changes from warm to full), the warm-buffered messages are flushed to the newly-created full connection.

If RTclient does not have a connection to RTserver, then TipcSrvCreate can be used to create a warm connection to RTserver. For example:

```

if (!TipcSrvCreate(T_IPC_SRV_CONN_WARM)) {
    /* error */
}

```

If RTclient has a full connection to RTserver, then TipcSrvDestroy can be used to leave a warm connection to RTserver. For example:

```
if (!TipcSrvDestroy(T_IPC_SRV_CONN_WARM)) {
    /* error */
}
```

Once RTclient destroys the full connection but leaves a warm connection, it can easily recreate a full connection to RTserver in the future and pick up where it left off. For more details on how TipcSrvDestroy works in this situation, see the man page for TipcSrvDestroy in the *TIBCO SmartSockets Application Programming Interface* reference.

Connection Status

There is some overlap between TipcSrvCreate and TipcSrvDestroy in the respect that both explicitly change the status of the connection to RTserver. The function TipcSrvGetConnStatus can be used to determine the status of the connection to RTserver. For example:

```
T_IPC_SRV_CONN_STATUS conn_status;

if (!TipcSrvGetConnStatus(&conn_status)) {
    /* error */
}
switch (conn_status) {
case T_IPC_SRV_CONN_NONE:
    TutOut("RTclient has no connection to RTserver.\n");
    break;
case T_IPC_SRV_CONN_WARM:
    TutOut("RTclient has a warm connection to RTserver.\n");
    break;
case T_IPC_SRV_CONN_FULL:
    TutOut("RTclient has a full connection to RTserver.\n");
    break;
default:
    /* error */
}
```

The main difference between TipcSrvCreate and TipcSrvDestroy is that TipcSrvCreate always moves the connection status closer to T_IPC_SRV_CONN_FULL, while TipcSrvDestroy moves the connection status the opposite direction towards T_IPC_SRV_CONN_NONE.

Continued Operation While RTserver is Down

As described in Automatically Reconnecting to RTserver on page 201, if an unrecoverable error occurs on the connection to RTserver, the full connection to RTserver is destroyed, but a warm connection is kept. If RTclient cannot successfully restart and reconnect to RTserver, it keeps the warm connection and continues operation. Depending on the settings of the Server_* options and the usage of the TipcSrv* functions, RTclient continues running and keeps trying to restart and reconnect to RTserver. When RTserver does come back up, then RTclient can create a full connection again.

RTclient-Specific Callbacks

In addition to all the callback types connections offer (see Callbacks on page 84 for details on connection callback types), there are several additional callback types RTclient can use including: server create callbacks, server destroy callbacks, and subject callbacks. Table 11 shows the callback type-specific data argument types for all RTclient callback types and the fields in those argument types.

Table 11 RTclient Callback Types

Callback Type	Type of Second Parameter to Callback Functions	Fields in This Type
server create	T_IPC_SRV_CREATE_CB_DATA	T_CB cb; T_IPC_SRV_CONN_STATUS old_conn_status; T_IPC_SRV_CONN_STATUS new_conn_status;
server destroy	T_IPC_SRV_DESTROY_CB_DATA	T_CB cb; T_IPC_SRV_CONN_STATUS old_conn_status; T_IPC_SRV_CONN_STATUS new_conn_status;
server names traverse	T_IPC_SRV_TRAVERSE_CB_DATA	T_CB cb; T_STR server_name; T_BOOL stop_traverse;
subject	T_IPC_SRV_SUBJECT_CB_DATA	T_CB cb; T_IPC_MSG msg;

Server Create Callbacks

Server create callbacks are called when an RTclient creates either a full or warm connection to RTserver. Server create callbacks are useful for creating other callbacks, such as connection process callbacks on the connection to RTserver, that cannot be created until RTclient has a connection to RTserver. If an RTclient creates and destroys its connection to RTserver many times, it should use server create callbacks to create the other callbacks needed in the connection to RTserver. For example:

```
/* ===== */
/*..cb_server_create -- server create callback */
static void T_ENTRY cb_server_create(
    T_IPC_CONN conn,
    T_IPC_SRV_CREATE_CB_DATA data,
    T_CB_ARG arg)
{
    T_IPC_MT mt; /* for creating callback */
    /* Create other callbacks if we did not have a warm connection. */
    if (data->old_conn_status == T_IPC_SRV_CONN_WARM) {
        TutOut("We already had a warm connection to RTserver, which ");
        TutOut("preserves all callbacks.\n");
        return; /* nothing to do */
    }

    /* A larger process would create many callbacks here. This */
    /* simple example only creates a connection default callback. */
    TutOut("Creating other callbacks.\n");

    /* default callback */
    if (TipcSrvDefaultCbCreate(cb_default, NULL) == NULL) {
        TutOut("Could not create default callback: error <%s>.\n",
            TutErrStrGet());
    }
} /* cb_server_create */
```

Server Destroy Callbacks

Server destroy callbacks are called when an RTclient destroys its connection to RTserver, leaving either a warm connection or no connection. Server destroy callbacks are the opposite of server create callbacks.

Server destroy callbacks do overlap slightly with connection error callbacks, because the standard connection error callback function `TipcCbSrvError` (see [Automatically Reconnecting to RTserver](#) on page 201) calls `TipcSrvDestroy`, which causes the server destroy callbacks to be executed. Server destroy callbacks are useful for RTclients that need to know when the connection to RTserver no longer has a valid socket, such as RTclients that are also Motif or Windows GUI programs. See the man page for `TipcSrvDestroyCbCreate` in the *TIBCO SmartSockets Application Programming Interface* reference for an example of how server destroy callbacks help in this situation.

Server Names Traverse Callbacks

Server names traverse callbacks are called when an RTclient traverses the Server_Names list to connect to an RTserver. The callback is executed before a connection to each entry in the list is attempted. There are many uses for this callback, and the argument you pass in TipcSrvTraverseCbCreate can be anything.

The callback structure includes two fields, `stop_traverse` and `server_name`. When you want the process to stop traversing the Server_Names list and fail the TipcSrvCreate, set `data->stop_traverse = T_TRUE`. If you know the thread ID of the receive thread, you can check to see if this is the receive thread and set `stop_traverse` to `TRUE` if it is not. You can also pass a structure containing the thread ID and other information allowing you to do sophisticated backoff algorithms. By setting your server reconnect delay to a low value, you can then control how long to wait in this callback.

Here are some considerations to keep in mind when using the callback:

- The callback is always called when a TipcSrvCreate is performed. If you don't want it invoked until after the initial connect is made, don't install it until after the first TipcSrvCreate is done.
- When an error occurs, this callback is called after the server reconnect delay has been performed. Set the Server_Reconnect_Delay option to a low value if you want the callback to perform the initial delay.
- The callback is called before trying to connect to any RTserver in the Server_Names list. It works even when you randomize the list.

This is a server names traverse callback used to print a message for each server name in the Server_Names list. It stops traversing the Server_Names list if a successful connection is made or connection attempts fail for the first three servers in the list:

```
#include <rtworks/ipc.h>

/*
===== */
/* ..my_server_traverse_cb - server name traversal callback */
void T_ENTRY my_server_traverse_cb
(
    T_IPC_CONN conn,
    T_IPC_SRV_TRAVERSE_CB_DATA data,
    T_CB_ARG arg
)
{
    T_INT4 *count = (T_INT4 *)arg;

    TutOut("Inside my_server_traverse_cb: server_name = %s\n",
           data->server_name);
```

```

    /* stop trying at 3 servers */
    if(*count >= 3) {
        TutWarning("Setting stop flag.\n");
        data->stop_traverse = T_TRUE;
    }
    *count += 1;
}

/*
=====*/
/* ..main - main program */
int main(int argc, char **argv) {

    T_INT4 trav_count = 0;

    /* create a callback to be called when traversing the server names list */
    if (TipcSrvTraverseCbCreate(my_server_traverse_cb,
                               (T_CB_ARG)&trav_count) == T_NULL) {
        /* error */;
    }

    TipcSrvCreate(T_IPC_SRV_CONN_FULL);

} /* main */

```

Subject Callbacks

Earlier in this document, it was described how to define callbacks to process messages based on the type of the message. For RTserver connections, subject callbacks are available which are actually a superset of connection process callbacks, such as execute this callback when a message of type *T* arrives with destination *S*. Subject callbacks give you all the power of process callbacks with additional filtering capabilities. You should consider using subject callbacks rather than process callbacks when processing messages from RTserver.

Rather than processing a message based only on its type, subject callbacks allow you to process a message based on both its destination (subject) and its type. With a subject callback, you can specify a separate function for each subject (or group) of subjects you wish to operate on. When a message arrives at the receiver for the specified subject and is ready to be processed, the callback is executed. Subject callbacks operate similar to process callbacks. To create a subject callback, you make a call to `TipcSrvSubjectCbCreate` as shown:

```
T_CB TipcSrvSubjectCbCreate(subject, mt, func, arg )
```

where:

subject is the destination you wish to specify the callback on and (NULL means any destination).

mt is the message type the callback should be applied to (NULL means any message type).

func is the callback function to be executed.

arg is an optional argument to pass into *func*.

Here are some examples of creating subject callbacks:

```
mt = TipcMtLookupByNum(T_MT_INFO);
```

```
/* Execute the function subj_cb for any message type which has a destination of "/stocks" */  
TipcSrvSubjectCbCreate("/stocks", NULL, subj_cb, NULL);
```

```
/* Execute the function subj_cb for any messages of type INFO, regardless of the destination */  
TipcSrvSubjectCbCreate(NULL, mt, subj_cb, NULL);  
/* or */  
TipcSrvSubjectCbCreate("/...", mt, subj_cb, NULL);
```

```
/* Execute the function subj_cb for any messages of type INFO, which have a destination of "/stocks" */  
TipcSrvSubjectCbCreate("/stocks", mt, subj_cb, NULL);
```

```
/* Execute the function subj_cb for messages of any type with any destination */  
TipcSrvSubjectCbCreate(NULL, NULL, subj_cb, NULL);
```

```
/* Execute the function default_subj_cb if there are no subject callbacks which match the message */  
TipcSrvSubjectDefaultCbCreate(default_subj_cb, NULL);
```

Wildcards can be used when specifying the *subject* argument when creating a subject callback. A `NULL` subject maps to the `"/..."` wildcard subject. When you register a wildcard subject the callback function will get executed for each incoming message whose destination matches the wildcard pattern. It is possible that a single message can invoke multiple callbacks if its destination matches multiple wildcards.



Connection process callbacks are separate from server subject callbacks in their priority sets. Do not mix connection process callbacks and server subject callbacks because the priorities between callback types are not honored.

Table 12 shows several examples of wildcarded subjects and message types and whether or not the subject callback gets executed.

Table 12 Subject Callback Execution

Subject	Message Type	When Callback Gets Executed
/stocks	NUMERIC_DATA	NUMERIC_DATA message is received with destination of /stocks.
/stocks	NULL	When any type of message is received with destination of /stocks.
/... or NULL	STRING_DATA	STRING_DATA message is received, regardless of the destination. This is similar to the connection process callback behavior except that subject callbacks are always called after connection process callbacks.
/... or NULL	NULL	When a message of any type with any destination is received (the default subject callback is never executed in this case).

Just as with process callbacks, you can define a default subject callback to be executed if no callback has been defined for a given subject. Default subject callbacks are only called when no other subject callback matches the incoming message criteria. Default subject callbacks are registered with the `TipcSrvSubjectDefaultCbCreate` function. If a subject callback is registered with a `NULL` subject and `NULL` message type, then default subject callbacks will never get executed.

As with other callback types, you can use the function `TutCbSetPriority` to set the priority of your subject callbacks.

Subject callbacks can be destroyed by looking up each callback and destroying it with the `TutCbDestroy` function. A convenience function, `TipcSrvSubjectCbDestroyAll`, has been created to facilitate destroying all subject callbacks including default subject callbacks.

See the man page for `TipcSrvSubjectCbCreate` and `TipcSrvSubjectDefaultCbCreate` in the *TIBCO SmartSockets Application Programming Interface* reference for an example of how subject callbacks work.

Example Using Subject Callbacks

This code shows the use of subject callbacks. Two different subject callbacks are defined in the program:

- `ProcessInfo` — executed when processing a message of type `INFO` with destination `EXAMPLE_SUBJECT`
- `ProcessNumData` — executed when processing a message of type `NUMERIC_DATA` with destination `EXAMPLE_SUBJECT`

Both callback functions simply access the fields and print out the contents of the message:

```
/* rtsubjcb.c - print out contents of messages processed via
subject callback */
/* $RTHOME/examples/smrtsock/manual/rtsubjcb.c */

#include <rtworks/ipc.h>
#include "rtclient.h"

/* ===== */
/*..ProcessInfo - callback for processing INFO messages published
to the EXAMPLE_SUBJECT subject */
static void T_ENTRY ProcessInfo(
    T_IPC_CONN conn,
    T_IPC_CONN_PROCESS_CB_DATA data,
    T_CB_ARG arg)
{
    T_IPC_MSG msg = data->msg;
    T_STR msg_text;

    TutOut("Entering ProcessInfo callback.\n");

    if (!TipcMsgSetCurrent(msg, 0)) {
        TutOut("Could not set current field of message: error <%s>.\n",
            TutErrStrGet());
        return;
    }
}
```

```

        if (!TipcMsgNextStr(msg, &msg_text)) {
            TutOut("Could not text from INFO message: error <%s>.\n",
                TutErrStrGet());
            return;
        }

        TutOut("Text from message = %s\n", msg_text);
    } /* ProcessInfo */
/* ===== */
/*..ProcessNumData - callback for processing NUMERIC_DATA messages
    published to the EXAMPLE_SUBJECT subject */
static void T_ENTRY ProcessNumData(
    T_IPC_CONN conn,
    T_IPC_CONN_PROCESS_CB_DATA data,
    T_CB_ARG arg)
{
    T_IPC_MSG msg = data->msg;
    T_STR name;
    T_REAL8 value;

    TutOut("Entering ProcessNumData callback.\n");
    if (!TipcMsgSetCurrent(msg, 0)) {
        TutOut("Could not set current field of message: error <%s>.\n",
            TutErrStrGet());
        return;
    }

    /* Access and print fields */
    while (TipcMsgNextStrReal8(msg, &name, &value)) {
        TutOut("Var Name = %s; Value = %s\n", name,
            TutRealToStr(value));
    } /* while */

    /* Make sure we reached end of message */
    if (TutErrNumGet() != T_ERR_MSG_EOM) {
        TutOut("Did not reach end of message: error <%s>.\n",
            TutErrStrGet());
    }
} /* ProcessNumData */

/* ===== */
int main(argc, argv)
int argc;
char **argv;
{
    T_OPTION option;
    T_IPC_MT mt;

```

```

/* Set the project name */
option = TutOptionLookup("project");
if (option == NULL) {
    TutOut("Could not look up option named project: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (!TutOptionSetEnum(option, EXAMPLE_PROJECT)) {
    TutOut("Could not set option named project: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* Connect to the RTserver */
TutOut("Creating a connection to RTserver.\n");
if (!TipcSrvCreate(T_IPC_SRV_CONN_FULL)) {
    TutOut("Could not connect to RTserver!\n");
    TutExit(T_EXIT_FAILURE);
} /* if */

/* Create subject callbacks to be executed when messages arrive
   with a given destination and given type */
TutOut("Creating subject callbacks.\n");

/* Subject callback for INFO messages with destination
   EXAMPLE_SUBJECT subject */
mt = TipcMtLookupByNum(T_MT_INFO);
if (mt == NULL) {
    TutOut("Could not lookup INFO message type: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (TipcSrvSubjectCbCreate(EXAMPLE_SUBJECT, mt,
    ProcessInfo, NULL)
    == NULL) {
    TutOut("Could not create ProcessInfo subject callback: ",
        "error <%s>.\n", TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* Subject callback for NUMERIC_DATA messages with destination
   EXAMPLE_SUBJECT subject */
mt = TipcMtLookupByNum(T_MT_NUMERIC_DATA);
if (mt == NULL) {
    TutOut("Could not lookup NUMERIC_DATA message type: ",
        "error ,<%s>.\n", TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (TipcSrvSubjectCbCreate(EXAMPLE_SUBJECT, mt,
    ProcessNumData, NULL)
    == NULL) {
    TutOut("Could not create ProcessNumData subject callback: ",
        "error <%s>.\n", TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

```

```

/* Start subscribing to the EXAMPLE_SUBJECT subject */
    TutOut("Start subscribing to the %s subject.\n",
EXAMPLE_SUBJECT);
    if (!TipcSrvSubjectSetSubscribe(EXAMPLE_SUBJECT, TRUE)) {
        TutOut("Could not subscribe to %s subject: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
/* Read and process all incoming messages */
    if (!TipcSrvMainLoop(T_TIMEOUT_FOREVER)) {
        TutOut("TipcSrvMainLoop failed: error <%s>.\n",
            TutErrStrGet());
    }
} /* main */

```

This code can be verified by first compiling, linking and running
\$RTHOME/examples/smrtsock/manual/rtsubjcb.c. Messages can be published
to it using the example program found in
\$RTHOME/examples/smrtsock/manual/rtsndsub.c.

Remote Procedure Calls

The section Remote Procedure Calls on page 147 describes how messages can be sent between processes to perform a remote procedure call. A remote procedure call (RPC) is a means for a process to execute a function in another process and wait for the result of the function call. This enables a request-reply communication model.

Normally when an RTclient publishes a message to a subject using TipcSrvMsgSend, the publishing RTclient continues and does not wait for the subscribing RTclients to receive the message and act on it. This normal mode of operation can be thought of as a one-to-many non-blocking RPC that may or may not return a result (depending on whether or not the receiving RTclients send back result messages).

The function TipcSrvMsgSendRpc performs a one-to-one blocking RPC that does return a result. One message is sent as the RPC call from the caller RTclient, and another message is sent back as the RPC result to the caller. The target RTclient must be prepared to receive the call message and send back the result message. TipcSrvMsgSendRpc uses a simple relationship between the call and result messages: the message type number of the result message must be one greater than the message type number of the call message.

See the reference information for TipcSrvMsgSendRpc in *TIBCO SmartSockets Application Programming Interface* for a code example of how to perform RPCs with other RTclients. TipcSrvMsgSendRpc cannot be used to achieve a one-to-many blocking RPC; this kind of RPC can be handled by sending the call message with TipcSrvMsgSend and then gathering all the necessary result messages with TipcSrvMsgSearchType.

Changing RTclient Options

Many options, such as Project, are only referenced when RTclient creates a connection to RTserver. One way to change these options dynamically is to destroy the connection to RTserver, keeping it warm, and then create a new RTserver connection using the warm information. Code like this can be used to change options like Project or Server_Names, the changes taking effect immediately:

```
/* destroy existing connection to RTserver, but keep it warm */
if (!TipcSrvDestroy(T_IPC_SRV_CONN_WARM)) {
    /* error */
}

/* change project */
TutCommandParseStr("setopt project new_project");
/* create a new connection to RTserver from warm info */
if (!TipcSrvCreate(T_IPC_SRV_CONN_FULL)) {
    /* error */
}
```

There are restrictions for changing certain RTclient options:

- Project

The Project option is set as read only while RTclient has a full connection to RTserver. This prevents confusion from resulting if this option is changed after the full connection is created.

- Unique_Subject and Default_Subject_Prefix

Because the Unique_Subject and Default_Subject_Prefix options may be used each time RTclient publishes a message, these options are set as read only while RTclient has a warm or full connection to RTserver. This prevents misconfiguration during normal publish-subscribe operation. The connection to RTserver must be fully destroyed (no warm connection can be left) and recreated before a change to the value of Unique_Subject or Default_Subject_Prefix can take effect.

Connecting to Multiple RTservers

Most RTclient applications require only one connection to an RTserver. The TipcSrv API uses a single global connection to an RTserver. For those rare applications whose architecture requires an RTclient to connect to multiple RTservers, those RTclients can use special multiple connections instead of a global connection.

Multiple RTserver connections use the TipcSrvConn API which allows distinct connections to multiple RTservers. Each connection has its own set of subscriptions and callbacks. Setting different option values for individual connections requires the use of named options, defined using the `setnopt` command or `TutOptionNamedLookup`, and the `TipcSrvConnCreateNamed` function. For information on the TipcSrvConn API, and the associated `TipcDispatcher` and `TipcEvent` APIs, see the *TIBCO SmartSockets Application Programming Interface*.

Certain SmartSockets commands cannot be used with multiple connections outside of CONTROL messages, as there is no way for the command processor to know for which connection the command is intended. The commands created with `TipcInitCommands` (`connect`, `disconnect`, `subscribe`, and `unsubscribe`) are applied to the global connection if processed via a direct API call. If these commands are received as CONTROL messages, they will act on the connection on which they were received.

Because file-based GMD, by default, creates a sub-directory named after the RTclient's unique subject, file conflicts can arise when different multiple RTserver connections sharing the same unique subject attempt to write to the same directory. To avoid this, specify a different GMD sub-directory for each connection using the `Server_Gmd_Dir_Name` option. For more information, see [File-based GMD and Connections to Multiple RTservers](#) on page 353.

A dispatcher is available to manage the incoming messages from multiple RTserver clouds. Using a dispatcher can increase the performance of your application. See [Using a Dispatcher](#) on page 249 for more information.



Using multiple connections is recommended when an RTclient:

- connects to two or more RTserver clouds, or
- connects to two or more projects in the same RTserver cloud.

In most other cases, multiple connections are unnecessary. If used unwisely, they can cause serious performance issues across your SmartSockets application.

Using a Dispatcher

An RTclient dispatcher manages tasks, determining when and in what order they execute. A dispatcher does one or both of these activities:

- manages the waiting process for messages from multiple RTserver connections

This is especially useful in single threaded RTclients. By adding one or more RTserver connections to a dispatcher with `TipcDispatcherSrvAdd`, a single `TipcDispatcherMainLoop` call is equivalent to calling `TipcSrvConnMainLoop` on each of the RTserver connections simultaneously.

- manages events originating within an RTclient, between RTclients, or between sockets

An event is an object registered in the dispatcher that manages it. The dispatcher determines when the event should execute. There are five kinds of events: connection, message, socket, timer, and user. See Events on page 254 for more information.

Types of Dispatchers

There are two kinds of dispatchers:

- a dispatcher to handle multiple tasks for a specified time interval

This kind of dispatcher is created with `TipcDispatcherCreate` and runs in the RTclient thread in which it is created. A call to `TipcDispatcherMainLoop` is required to run this dispatcher. `TipcDispatcherMainLoop` also defines the time interval for the dispatcher to run before exiting its main loop.

While the dispatcher is running, it checks its registered events to determine if any should be executed, and waits for messages from RTservers. This is the most common type of dispatcher to use for RTclients. See Single Threaded RTclients on page 251 for more information.

- a dispatcher to run continuously in a detached thread

This kind of dispatcher is created in a separate thread with `TipcDispatcherCreateDetached`. A detached dispatcher runs indefinitely in the background and can only be stopped with `TipcDispatcherDestroy`. See Example 22 on page 265 for an example of using this kind of dispatcher.

This table summarizes the SmartSockets functions used for a dispatcher:

Function Name	Purpose
TipcDispatcherCreate	Creates a dispatcher to manage incoming messages on an RTserver connection. The dispatcher can also manage connection, message, socket, timer, and user events.
TipcDispatcherCreateDetached	Creates a dispatcher in a detached thread, which is a thread running on its own in the background. The dispatcher runs continuously until destroyed with TipcDispatcherDestroy.
TipcDispatcherDestroy	Removes a dispatcher created with TipcDispatcherCreate or TipcDispatcherCreateDetached.
TipcDispatcherDispatch	Provides an alternate method of waiting for messages or managing events, as opposed to using TipcDispatcherMainLoop. See How a Dispatcher Executes Events on page 255 for more information.
TipcDispatcherMainLoop	Controls how long the dispatcher dispatches for messages or manages events. After the defined interval of time elapses, TipcDispatcherMainLoop exits. The dispatcher does not function again until TipcDispatcherMainLoop is called again.
TipcDispatcherSrvAdd	Adds an RTserver connection to a dispatcher.
TipcDispatcherSrvRemove	Removes an RTserver connection from a dispatcher.

See the *TIBCO SmartSockets Application Programming Interface* for more information on these functions.

Single Threaded RTclients

When a single threaded RTclient has connections to multiple RTservers, using a dispatcher to wait for messages can greatly increase the performance of the RTclient.

What Happens Without a Dispatcher?

Without a dispatcher, an RTclient's normal procedure of waiting for a message from an RTserver, reading the message, and processing the message, is complicated by the fact that more than one RTserver is sending messages to the RTclient. The wait-read-process cycle is multiplied by the number of RTserver connections, which can easily slow down the RTclient because the wait-read-process steps for each RTserver connection are performed sequentially.

For example, an RTclient has two RTserver connections, one for RTserver A and one for RTserver B. Here is a summary of how some of the wait-read-process steps might decrease the performance of the RTclient:

1. Wait for messages from RTserver A until a message arrives.
2. When a message arrives, read it from RTserver A.
3. Send the message to the RTclient's connection for RTserver A for processing and wait for processing to complete.
4. Wait for messages from RTserver B until a message arrives.
5. When a message arrives, read it from RTserver B.
6. Send the message to the RTclient's connection for RTserver B for processing and wait for processing to complete.
7. Wait for messages from RTserver A, and so on.

Except for reading a message, every step offers the potential to block all other processing until it is completed. If 200 messages a second flow into this RTclient from the RTservers, you can see how performance would be slowed down as time spent waiting for and processing a message accumulates.

How a Dispatcher Improves Performance

Using a dispatcher causes the wait-read-process steps to be multiplexed in such a way that long periods of time associated with the wait steps are greatly decreased.

For example, you have an RTclient connected to RTserver A and RTserver B. Here is a summary of how some of the wait-read-process steps might be managed by the dispatcher:

1. Wait for messages from RTserver A and RTserver B. The first message to arrive is from RTserver B.
2. Read the message from RTserver B.
3. Send the message to the RTclient's connection for RTserver B for processing and wait for processing to complete.
4. Wait for messages from RTserver A or B as done in Step 1. The second message to arrive is from RTserver A.
5. Send this message to the RTclient's connection for RTserver A for processing, and wait for processing to complete.
6. Wait for a message from RTserver A or B, and so on.

With a dispatcher handling the waiting steps more efficiently, the overall performance of the RTclient is increased. Further increase in performance is possible if you design your RTclient to process each message in the quickest way.

Using a Dispatcher

Adding RTserver connections to a dispatcher and running the dispatcher's main loop is equivalent to calling the main loop function of all the RTserver connections at the same time.

The code shown in Example 20 and Example 21 result in the same behavior: waiting for a message on one RTserver connection.

Example 20 Waiting for Messages Without a Dispatcher

```
/* Create an RTserver connection. */
TipcSrvCreate(T_IPC_SRV_CONN_FULLL);

/* Register a default callback to wait for messages. */
TipcSrvDefaultCbCreate(defCbFunc, T_NULL);

/* Receive messages from the RTserver. */
TipcSrvMainLoop(T_TIMEOUT_FOREVER);
```

Example 21 Waiting for RTserver Messages With a Dispatcher

```

T_IPC_SRV srv;
T_IPC_DISPATCHER disp;

/* Create an RTserver connection. */
TipcSrvCreate(T_IPC_SRV_CONN_FULL);

/* Register a default callback to wait for messages. */
TipcSrvDefaultCbCreate(defCbFunc, T_NULL);

/* Create a dispatcher and add the RTserver connection to it. */
disp = TipcDispatcherCreate();
TipcSrvGetSrv(&srv);
TipcDispatcherSrvAdd(disp, srv);

/* Receive messages from the RTserver. */
TipcDispatcherMainLoop(disp, T_TIMEOUT_FOREVER);

```

The difference between the two examples is that more connections to more RTservers can be added to the dispatcher `disp` in Example 21 with `TipcDispatcherSrvAdd`. The dispatcher could also manage various events affecting the RTclient. Waiting for messages from all the RTservers and dealing with events only requires a single main loop, the dispatcher's.

Events

An event, an object registered in the dispatcher that manages it, is a way to:

- monitor the availability of a socket or connection for read and write operations
- process a message based on the message’s subject or type
- communicate between the threads of an RTclient
- cause a function of your RTclient to execute repeatedly at a set interval of time

This table summarizes the types of events managed by a dispatcher:

Event Type	Description
Connection	<p>A connection event executes when an RTclient connection is available for reading or writing a message to another RTclient. A connection event is similar to a socket event except the sockets are SmartSockets connections between RTclients.</p> <p>See Connection Events on page 256 for more information.</p>
Message	<p>A message event executes when an incoming message matches the message type or subject defined for the event.</p> <p>See Message Events on page 258 for more information.</p>
Socket	<p>Socket events are used when integrating other socket-based software into a dispatcher’s main loop. A socket event executes when a socket is available for sending or receiving a message.</p> <p>See Socket Events on page 260 for more information.</p>
Timer	<p>A timer event executes at a time interval you define. The event usually repeats but can be executed only once.</p> <p>See Timer Events on page 262 for more information.</p>
User	<p>A user event is useful for inter-thread communication. By running a dispatcher for each thread, user events can be sent between dispatchers in a manner that does not disrupt a thread’s processing.</p> <p>See User Events on page 263 for more information.</p>

How a Dispatcher Executes Events

It is the dispatcher's role to recognize when an event's execution criteria is met. When the criteria is met, it triggers the dispatcher to execute the callback function associated with the event. A callback function always executes in the thread that dispatches the event. You specify the thread in which you want the callback function executed by passing a dispatcher's identifier in the `TipcEventCreate*` function at the time an event is registered. An event's callback function should not run for a long time because this delays the execution of other events waiting to be dispatched.

For example, a connection event is triggered whenever a connection is available for a read operation. When the dispatcher determines a connection is available for reading, it executes the connection event's callback function to do the actual work of reading the incoming message from the connection.

The typical way to run a dispatcher involves calling `TipcDispatcherMainLoop` to run the dispatcher for a specified period of time. During the time period, the dispatcher manages the wait-read-process steps for messages from RTservers, and determines if any of its events were triggered for execution. After the time period elapses, the dispatcher exits from its main loop and does not run until `TipcDispatcherMainLoop` is called again.

Another way to run a dispatcher is to call `TipcDispatcherDispatch`. This function also defines a time period for the dispatcher to run, but the dispatcher exits its main loop as soon as it handles one or more events. It only runs for its entire time period if no messages arrive or there are no events to execute.

Examples of Using `TipcDispatcherDispatch`:

1. `TipcDispatcherDispatch` is called to run a dispatcher for 3 seconds. The dispatcher determines after 1 second that a timer event is triggered. After invoking the timer event, the dispatcher exits rather than running for the remaining 2 seconds of its time period.
2. `TipcDispatcherDispatch` is called to run a dispatcher for 5 seconds. The dispatcher determines after 2 seconds that a message has arrived from an RTserver, a message event is triggered, and a timer event is triggered. After reading the message and invoking the two events, the dispatcher exits rather than running for the remaining 3 seconds of the time period.
3. `TipcDispatcherDispatch` is called to run a dispatcher for 2 seconds. No RTserver messages arrive and no events are triggered during the time period. The dispatcher runs for 2 seconds before exiting its main loop.

Connection Events

Connection events are a special method of communicating between SmartSockets connections without routing the messages through an RTserver. This kind of connection is referred to as the peer-to-peer model.

A connection event is executed by a dispatcher when the connection associated with the event is ready for a read or write operation. When you register a connection event, you must specify which operation triggers the event with a check mode value of `T_IO_CHECK_READ` or `T_IO_CHECK_WRITE`.

Read-Mode Example

These steps take place to implement a connection event for read-mode in an RTclient:

1. The RTclient registers a connection event of check mode `T_IO_CHECK_READ` with the dispatcher that monitors the connection.
2. When the connection is available to accept an incoming message in its socket, the dispatcher executes the connection event.
3. The callback function of the connection event reads the message from the connection's socket.

Write-Mode Example

These steps take place to implement a connection event for write-mode in an RTclient:

1. The RTclient registers a connection event of check mode `T_IO_CHECK_WRITE` mode with the dispatcher that monitors the connection.
2. When the connection is available to accept an outgoing message on its socket, the dispatcher executes the connection event.
3. The callback function of the connection event writes the message to the connection's socket.

This table summarizes the SmartSockets functions used for SmartSockets connection events:

Function Name	Purpose
TipcEventCreateConn	Registers a SmartSockets connection event with a dispatcher. The event's execution is triggered when a message can be written to a connection (T_IO_CHECK_WRITE mode), or a message can be read from a connection (T_IO_CHECK_READ mode).
TipcEventDestroy	Removes a connection event from a dispatcher.
TipcEventGetCheckMode	Returns the triggering mode for a connection event (T_IO_CHECK_READ or T_IO_CHECK_WRITE).
TipcEventGetConn	Returns the identifier of the connection for a connection event.
TipcEventGetDispatcher	Returns the identifier of the dispatcher where a connection event is registered.
TipcEventGetType	Returns the type of an event. When used for a connection event, the returned value is T_IPC_EVENT_CONN.

See the *TIBCO SmartSockets Application Programming Interface* for more information on these functions.

Message Events

With message events, the processing of a message can easily be spread across multiple threads based on the message's type or subject.

When using message events, a message travels from the sending RTclient through the RTserver to the receiving RTclient's connection or dispatcher. The RTclient's connection or dispatcher, whichever is waiting for messages, sends the message to the appropriate thread's dispatcher.

The receiving dispatcher determines if this message matches any of its registered message types or subjects. Upon a match, the dispatcher immediately triggers the execution of a callback function to process the message.



Messages must not be destroyed within a message event's callback function. SmartSockets automatically destroys a message once all the events have received it.

See the Multiple Thread Example with Timer and Message Events on page 264 for an example of using message events.

This table summarizes the SmartSockets functions used for message events:

Function Name	Purpose
TipcEventCreateMsg	Registers a message event with a dispatcher. The event's execution is triggered by the subject of the incoming message.
TipcEventCreateMsgType	Registers a message event with a dispatcher. The event's execution is triggered by the type of incoming message.
TipcEventDestroy	Removes a message event from a dispatcher.
TipcEventGetDispatcher	Returns the identifier of the dispatcher where a message event is registered.
TipcEventGetType	<p>Returns the type of an event. When used for a message event, the returned value is:</p> <ul style="list-style-type: none"> • T_IPC_EVENT_MSG if the message event is triggered by a subject, or • T_IPC_EVENT_MSG_TYPE if the message event is triggered by a message's type

See the *TIBCO SmartSockets Application Programming Interface* for more information on these functions.

Socket Events

Socket events, similar to connection events, are events that signal a message can be sent or received between the sockets of a client and another vendor's server. The client can be an RTclient connected to an RTserver and another vendor's server, or it can be a client only connected to another vendor's server. In either case, the client uses the SmartSockets dispatcher to handle the execution of the socket event.

A socket event is executed by the dispatcher when the vendor's socket is ready for a read or write operation. When you register a socket event, you must specify which operation triggers the event with a check mode value of `T_IO_CHECK_READ` or `T_IO_CHECK_WRITE`.

Any vendor's software that supports the ANSI standard for socket architecture can be used in a socket event. However, the socket must be in non-blocking mode to support the asynchronous model of SmartSockets. A write event is triggered when one or more bytes can be written to the vendor's socket. A read event is triggered for these conditions:

- one or more bytes can be read from the vendor's socket
- a socket error occurred, such as the vendor's socket peer closed the connection

This table summarizes the SmartSockets functions used for sockets events:

Function Name	Purpose
TipcEventCreateSocket	Registers a socket event with a dispatcher. The event's execution is triggered when a message can be written to the socket (T_IO_CHECK_WRITE mode), or a message can be read from the socket (T_IO_CHECK_READ mode).
TipcEventDestroy	Removes a socket event from a dispatcher.
TipcEventGetCheckMode	Returns the triggering mode for a socket event (T_IO_CHECK_READ or T_IO_CHECK_WRITE).
TipcEventGetDispatcher	Returns the identifier of the dispatcher where a socket event is registered.
TipcEventGetSocket	Returns the identifier of the socket for a socket event.
TipcEventGetType	Returns the type of an event. When used for a socket event, the returned value is T_IPC_EVENT_SOCKET.

See the *TIBCO SmartSockets Application Programming Interface* for more information on these functions.

Timer Events

Timer events originate within an RTclient. After a defined number of seconds has elapsed, the dispatcher triggers the execution of the timer event. A timer event is executed repeatedly unless it is destroyed within the timer event's callback function after the first time it executes.

See the Multiple Thread Example with Timer and Message Events on page 264 for an example of using timer events.

This table summarizes the SmartSockets functions used for timer events:

Function Name	Purpose
TipcEventCreateTimer	Registers a timer event with a dispatcher. The event's execution is triggered every time a set number of seconds, defined by TipcEventSetInterval, has elapsed.
TipcEventDestroy	Removes a timer event from a dispatcher.
TipcEventGetDispatcher	Returns the identifier of the dispatcher where a timer event is registered.
TipcEventGetInterval	Returns the number of seconds set for a timer event.
TipcEventGetType	Returns the type of an event. When used for a timer event, the returned value is T_IPC_EVENT_TIMER.
TipcEventSetInterval	Defines the number of seconds to wait between executing the timer event.

See the *TIBCO SmartSockets Application Programming Interface* for more information on these functions.

User Events

An RTclient communicates between its threads with user events. A dispatcher always executes a user event immediately when the event becomes the next in line for the dispatcher's attention. Unlike the other kinds of events, there is no triggering condition to be met before it is executed.

Unlike connection, message, socket, and timer events, which persist until removed with `TipcEventDestroy`, a user event is temporary. A user event must be registered each time it is required because after the user event executes, it is automatically destroyed.

Life Cycle of One User Event:

1. Thread 1 registers a user event in Thread 2's dispatcher. The user event becomes the third item in the dispatcher's queue, behind a connection event (`CERead`) and a timer event that executes every 5 seconds (`TE-5`).
2. During the current time period for `TipcDispatcherMainLoop` to run, the dispatcher is able to check all three events in its queue. A trigger has not occurred for `CERead`, so the dispatcher goes to the next event in line, `TE-5`. Five seconds have elapsed since the last time `TE-5` ran so the dispatcher directs it to run again.
3. After `TE-5` executes, the next event in line, `UE-T1`, is the user event registered by Thread 1. The dispatcher executes it immediately because the event does not require a trigger. `UE-T1` is destroyed upon its completion.

This table summarizes the `SmartSockets` functions used for user events:

Function Name	Purpose
<code>TipcEventCreate</code>	Registers a user event with a dispatcher.
<code>TipcEventGetData</code>	The user-defined data associated with the user event when it was registered with the dispatcher.
<code>TipcEventGetDispatcher</code>	Returns the identifier of the dispatcher where a user event was registered.
<code>TipcEventGetType</code>	Returns the type of an event. When used for a user event, the returned value is <code>T_IPC_EVENT_USER</code> .

See the *TIBCO SmartSockets Application Programming Interface* for more information on these functions.

Multiple Thread Example with Timer and Message Events

The following example uses timer and message events. Here is what takes place:

- Three threads are created in RTclient A (Example 22), each running a dispatcher.
- The main thread in RTclient A connects to an RTserver and adds the connection to its dispatcher.
- Two timer events are registered with the main thread's dispatcher. One timer event sends a T_MT_CLIENT_REQUEST message to RTclient B every second. The other timer event sends a T_MT_SERVER_REQUEST message to RTclient B every 5 seconds. These request messages result in response messages from RTclient B of type T_MT_CLIENT_RESPONSE or T_MT_SERVER_RESPONSE.

RTclient B is presented in Example 23 on page 268.

- RTclient A has two detached threads, each with its own dispatcher. A message event is registered with each of these dispatchers. One event processes message type T_MT_CLIENT_RESPONSE and the other processes message type T_MT_SERVER_RESPONSE. The message events are triggered by an incoming message's type.
- When the main thread's dispatcher receives a message from RTclient B of message type T_MT_CLIENT_RESPONSE, it sends the message to the appropriate dispatcher. The receiving dispatcher executes the message event to process the message. The same action occurs when message type T_MT_SERVER_RESPONSE is received.

Even if a response message takes a long time to process by one message event, the main thread is not blocked.

Example 22 Code for RTclient A

```

#include <rtworks/ipc.h>

#define T_MT_CLIENT_REQUEST    101
#define T_MT_CLIENT_RESPONSE   201

#define T_MT_SERVER_REQUEST    102
#define T_MT_SERVER_RESPONSE   202

/*-----*/
static void T_ENTRY clientRequestEventTimerFunc
(
    T_IPC_EVENT event,
    T_IPC_EVENT_DATA data,
    T_PTR arg
)
{
    /*-----
    * This is the callback function invoked whenever the associated timer event is triggered. The timer event sends
    * a type T_MT_CLIENT_REQUEST message to RTclient B. A message event in RTclient A processes the
    * type T_MT_CLIENT_RESPONSE message returned from RTclient B in response to the T_MT_CLIENT_
    * REQUEST message.
    *-----*/

    TutOut("%s: CLIENT REQUEST - sending\n", TutGetWallTimeStr());

    /*-----
    * Send the RTclient request message.
    *-----*/
    TipcSrvMsgWrite("/request", TipcMtLookupByNum(T_MT_CLIENT_REQUEST),
                    T_TRUE, T_NULL);
    TipcSrvFlush();
} /* clientRequestEventTimerFunc */

/*-----*/
static void T_ENTRY serverRequestEventTimerFunc
(
    T_IPC_EVENT event,
    T_IPC_EVENT_DATA data,
    T_PTR arg
)
{
    /*-----
    * This is the callback function invoked whenever the associated timer event is triggered. It sends a type
    * T_MT_SERVER_REQUEST message to RTclient B. A message event in RTclient A processes the
    * type T_MT_SERVER_RESPONSE message returned from RTclient B in response to the T_MT_SERVER_
    * REQUEST message.
    *-----*/

    TutOut("%s: SERVER REQUEST - sending\n", TutGetWallTimeStr());

    /*-----

```

```

* Send the server request message.
*/-----
TipcSrvMsgWrite("/request", TipcMtLookupByNum(T_MT_SERVER_REQUEST),
               T_TRUE, T_NULL);
TipcSrvFlush();

} /* serverRequestEventTimerFunc */
/*-----*/
static void T_ENTRY clientResponseEventMsgTypeFunc
(
    T_IPC_EVENT event,
    T_IPC_EVENT_DATA data,
    T_PTR arg
)
{
    /* -----
    * This is the callback function invoked whenever the associated message event is triggered. It processes a type
    * T_MT_CLIENT_RESPONSE message from RTclient B.
    */-----

    TutOut("%s: CLIENT RESPONSE - processing\n", TutGetWallTimeStr());
} /* clientResponseEventMsgTypeFunc */

/*-----*/
static void T_ENTRY serverResponseEventMsgTypeFunc
(
    T_IPC_EVENT event,
    T_IPC_EVENT_DATA data,
    T_PTR arg
)
{
    /* -----
    * This is the callback function invoked whenever the associated message event is triggered. It processes a
    * type T_MT_SERVER_RESPONSE message from RTclient B.
    */-----

    TutOut("%s: SERVER RESPONSE - processing\n", TutGetWallTimeStr());
} /* serverResponseEventMsgTypeFunc */

/*-----*/
int main
(
    int argc,
    char **argv
)
{
    T_IPC_SRV srv;
    T_IPC_DISPATCHER main_disp;
    T_IPC_DISPATCHER client_response_disp;
    T_IPC_DISPATCHER server_response_disp;

```

```

/* -----
 * Make the SmartSockets libraries thread-safe.
 */ -----
TipcInitThreads();

/* -----
 * Parse the command file if available.
 */ -----
TutCommandParseFile("request.cm");

/* -----
 * Create the message types.
 */ -----
TipcMtCreate("client request", T_MT_CLIENT_REQUEST, "verbose");
TipcMtCreate("client response", T_MT_CLIENT_RESPONSE, "verbose");

TipcMtCreate("server request", T_MT_SERVER_REQUEST, "verbose");
TipcMtCreate("server response", T_MT_SERVER_RESPONSE, "verbose");

/* -----
 * Create a connection to the RTserver.
 */ -----
TipcSrvCreate(T_IPC_SRV_CONN_FULL);

/* -----
 * Get the connection's T_IPC_SRV object.
 */ -----
TipcSrvGetSrv(&srv);

/* -----
 * Create a dispatcher.
 */ -----
main_disp = TipcDispatcherCreate();

/* -----
 * Add the connection for the RTserver to the dispatcher.
 */ -----
TipcDispatcherSrvAdd(main_disp, srv);

/* -----
 * Add two timer events to the dispatcher, one for each type of request message
 */ -----
TipcEventCreateTimer(main_disp,
                    1.0,
                    clientRequestEventTimerFunc,
                    T_NULL);
TipcEventCreateTimer(main_disp,
                    5.0,
                    serverRequestEventTimerFunc,
                    T_NULL);

```

```

/* -----
 * Create two dispatchers, each running in their own thread, to process two types of response messages.
 * Response messages are read from the RTserver connection in the main thread and handed off to one of the two
 * background threads for processing. This allows for concurrent processing of messages based on message
 * type, T_MT_CLIENT_RESPONSE or T_MT_SERVER_RESPONSE. A similar setup could be used
 * to processes messages based on subject.
 */ -----
client_response_disp = TipcDispatcherCreateDetached();
TipcEventCreateMsgType(client_response_disp,
                       srv,
                       TipcMtLookupByNum(T_MT_CLIENT_RESPONSE),
                       clientResponseEventMsgTypeFunc,
                       T_NULL);

server_response_disp = TipcDispatcherCreateDetached();
TipcEventCreateMsgType(server_response_disp,
                       srv,
                       TipcMtLookupByNum(T_MT_SERVER_RESPONSE),
                       serverResponseEventMsgTypeFunc,
                       T_NULL);

/* -----
 * Dispatch the timer events and wait for messages from the RTserver.
 */ -----
TipcDispatcherMainLoop(main_disp, T_TIMEOUT_FOREVER);

TutExit(T_EXIT_SUCCESS);
} /* main */

```

Example 23 Code for RTclient B

```

#include <rtworks/ipc.h>

#define T_MT_CLIENT_REQUEST    101
#define T_MT_CLIENT_RESPONSE   201

#define T_MT_SERVER_REQUEST    102
#define T_MT_SERVER_RESPONSE   202

/* ----- */
static void T_ENTRY clientRequestCbFunc
(
    T_IPC_CONN conn,
    T_IPC_CONN_PROCESS_CB_DATA data,
    T_CB_ARG arg
)
{
    T_STR sender;

    TipcMsgGetSender(data->msg, &sender);

    TutOut("%s: CLIENT REQUEST - received\n", TutGetWallTimeStr());
}

```

```

/* -----
 * Respond to the RTclient request message with an RTclient response message.
 */ -----
TipcSrvMsgWrite(sender, TipcMtLookupByNum(T_MT_CLIENT_RESPONSE),
                T_TRUE, T_NULL);
TipcSrvFlush();
} /* clientRequestCbFunc */

/* ----- */
static void T_ENTRY serverRequestCbFunc
(
    T_IPC_CONN conn,
    T_IPC_CONN_PROCESS_CB_DATA data,
    T_CB_ARG arg
)
{
    T_STR sender;

    TipcMsgGetSender(data->msg, &sender);

    TutOut("%s: SERVER REQUEST - received\n", TutGetWallTimeStr());

/* -----
 * Respond to the RTserver request message with an RTserver response message.
 */ -----
    TipcSrvMsgWrite(sender, TipcMtLookupByNum(T_MT_SERVER_RESPONSE),
                    T_TRUE, T_NULL);
    TipcSrvFlush();
} /* serverRequestCbFunc */
/* ----- */
int main
(
    int argc,
    char **argv
)
{
    /* -----
     * Parse the command file if available.
     */ -----
    TutCommandParseFile("response.cm");

    /* -----
     * Create the message types.
     */ -----
    TipcMtCreate("client request", T_MT_CLIENT_REQUEST, "verbose");
    TipcMtCreate("client response", T_MT_CLIENT_RESPONSE, "verbose");

    TipcMtCreate("server request", T_MT_SERVER_REQUEST, "verbose");
    TipcMtCreate("server response", T_MT_SERVER_RESPONSE, "verbose");

    /* -----
     * Create a connection to the RTserver.
     */ -----
    TipcSrvCreate(T_IPC_SRV_CONN_FULL);

```

```

/* -----
 * Subscribe to /request messages.
 */ -----
TipcSrvSubjectSetSubscribe("/request", T_TRUE);
TipcSrvFlush();

/* -----
 * Add two callbacks, one for each type of request messages.
 */ -----
TipcSrvProcessCbCreate(TipcMtLookupByNum(T_MT_CLIENT_REQUEST),
                      clientRequestCbFunc,
                      T_NULL);
TipcSrvProcessCbCreate(TipcMtLookupByNum(T_MT_SERVER_REQUEST),
                      serverRequestCbFunc,
                      T_NULL);

/* -----
 * Process messages from the RTserver.
 */ -----
TipcSrvMainLoop(T_TIMEOUT_FOREVER);

TutExit(T_EXIT_SUCCESS);
} /* main */

```


Message Compression

SmartSockets allows messages to be compressed before they are sent. There are three ways to compress a message:

- Compress on a message-by-message basis.
- Compress all messages of a certain message type.
- Compress all messages sent over a connection. This is called connection level compression.

Compression and decompression of messages is CPU time intensive, and should only be used when:

- message size is very large
- sending string-based or text messages, such as XML documents
- sending messages over a WAN, to conserve bandwidth

The compression library provided by SmartSockets is ZLIB (version 1.1.4). ZLIB is a general-purpose data compression library that provides in-memory compression and decompression, including integrity checks of the decompressed data. For more information on ZLIB, see the official ZLIB website, at <http://www.gzip.org/zlib/>.

To read a compressed message, the receiver must use the same compression library as was used to compress the message. If a message consumer receives a compressed message that it cannot decompress, usually through a `TipcMsgNext*` function call, the function fails with error:

- `T_ERR_DOESNT_EXIST` if the compression library could not be loaded, or
- `T_ERR_VAL_INVALID` if the compression libraries do not match.

Compressing by Message Type

Message type compression allows you to specify that any message of a particular type, such as all `STRING_DATA` messages, be compressed before it is sent to RTserver. Only the SmartSockets message payload, or user-data, is compressed.

The message is not decompressed until the first attempt is made to access the payload. In other words, the message remains compressed until the message receiver attempts to read the message. RTserver does not decompress the message; compression and decompression only takes place end-to-end, not at every hop.



If compression is enabled for a message but SmartSockets cannot load the compression library, it prints a warning and sends the message uncompressed.

Compression does not always result in a smaller payload. When compression yields a larger payload, SmartSockets prints a warning and sends the message uncompressed.

Enable or disable message type compression with the function `TipcMtSetCompression`. When compression is set, you can change the compression level with the `Compression_Args` option. For more information on `TipcMtSetCompression`, see the *TIBCO SmartSockets Application Programming Interface*.

Compressing a Single Message

You can always override the default message type compression setting for a given message with the function `TipcMsgSetCompression`. If compression for a message type is enabled but you wish to send a message uncompressed, use `TipcMsgSetCompression` to change the compression setting for that message only.

For more information on `TipcMsgSetCompression`, see the *TIBCO SmartSockets Application Programming Interface*.

Compressing at the Connection Level

Connection level compression causes all data sent across a connection (peer-to-peer, RTclient-RTserver, or RTserver-RTserver) to be compressed at the sender and decompressed at the receiver. Connection level compression is only available on point-to-point connections such as TCP, and is not available for PGM.

Connection level compression, as implemented by ZLIB, builds a rich dictionary and, over time, achieves higher compression ratios. The more messages you send over a connection, the better compression becomes. With connection level compression, the message header and payload are both compressed, and messages are decompressed at every hop.

Connection level compression is enabled with the SmartSockets Compression option. You can configure the compression setting by using an extended logical connection name (LCN), and with the options `Compression_Args`, `Compression_Name`, and `Compression_Stats`. Both sides of the connection must have the same `Compression` and `Compression_Name` option values in order for a connection to be made.

Using the Logical Connection Name

Connection level compression can be fully enabled and configured with SmartSockets options. However, you can also change compression settings with an extended logical connection name (LCN). Compression settings given in the LCN override options settings. For example, you can use the `Compression` option to enable compression across all connections by default, then use an extended LCN to disable compression for a specific connection.

LCNs are described in more detail in *Logical Connection Names* on page 101 and also in *Logical Connection Names for RT Processes* on page 192.

This section describes only the extended portion that is used to configure connection level compression.

The extended logical connection name has the form:

protocol:node:address[?*name=value*[&*name=value*]]

where:

protocol is the IPC protocol type

node is a computer node name

address is a protocol-specific IPC location, such as a TCP port number

name represents the connection property to be set. Properties that configure connection level compression are:

- *compression* — enables or disables compression for the connection. If this property is omitted, the compression setting defined by the SmartSockets Compression option is used.
- *compression_name* — specifies the compression library to use. If this property is omitted then the compression name defined by the SmartSockets Compression_Name option is used.
- *compression_args* — lists the arguments to pass to the compression library. If this property is omitted then the compression arguments defined by the SmartSockets Compression_Args option is used.

value is the value assigned to the connection property. Supported values for connection level compression properties are:

- if *name* is *compression*, use TRUE to enable compression; FALSE otherwise.
- if *name* is *compression_name*, use ZLIB. The compression library provided by SmartSockets is ZLIB.
- if *name* is *compression_args*, the value is determined by the compression library. For the ZLIB compression library, this is an integer value from 1 to 9.

If no compression settings are specified in the LCN, SmartSockets compression is determined by the Compression option. By default, compression is disabled.

Including the *compression=true* property in the LCN enables compression for the connection. This is an example of enabling compression for one of two connections in the RTserver:

```
setopt Conn_Names tcp:_node:5998, tcp:_node:5999?compression=true
```

You can also disable compression for a connection:

```
setopt Compression true
setopt Conn_Names tcp:_node:5998?compression=false, tcp:_node:5999
```

Both sides of the connection must have the same *compression* and *compression_name* property values in order for a connection to be made.

Security

SmartSockets offers username- and password-based security. This allows the RTserver to authenticate and authorize an RT process by:

- verifying that the connecting RT process is who it claims to be
- determining whether the connecting RT process is authorized to perform an action or access a resource

This is offered with Basic Security.

To enable Basic Security, edit the command file `rtserver.cm`, adding this line to set the SmartSockets option `Sm_Security_Driver` to `basic`:

```
setopt sm_security_driver basic
```

This command instructs the server to load the security command file `sdbasic.cm`. The file `sdbasic.cm` sets options for the Basic Security driver, such as the tracing level for logging security activity.

When Basic Security loads, it also reads and loads the access control list (ACL) configuration files. The ACL files are cached for the amount of time set in the `Sd_Basic_Acl_Timeout` option, then read again. If you make any changes to the ACL files after loading Basic Security, they do not take effect until the time specified in the `Sd_Basic_Acl_Timeout` option has passed, at which time the ACL files are reloaded.

Basic Security

SmartSockets Basic Security allows the RTserver to authenticate a user by requiring a username and password before accepting a connection. RTserver can also restrict access to resources by authorizing only certain users to access them. Basic Security uses a permission scheme to manage authorization. By setting permissions, you can control:

- server connections — which RT processes, or users, can connect to RTserver
- subject subscriptions — which users can subscribe to a subject
- subject publications — which users can publish to a subject

Basic Security can also provide auditing information, written to file or standard output (`stdout`). This information gives details on which users passed or failed authentication and which users were granted or denied access to resources. See `Sd_Basic_Trace_Level` on page 557 for more information.

Basic Security is managed through access control lists (ACLs). The ACL configuration files contain the usernames and passwords, group definitions, and permissions. The ACL files are stored in a file local to the machine which RTserver runs on, in the directory `$RTHOME/acl`. There are three configuration files:

- `users.cfg` — lists the username and password for each user authorized to connect to RTserver
- `groups.cfg` — defines groups, or categories, of users
- `acl.cfg` — sets the user and group permissions

To change the configuration files, you must edit them directly.



The ACL files are cached for the amount of time set in the `Sd_Basic_Acl_Timeout` option, then read again. If you make any changes to the ACL files after loading Basic Security, they do not take effect until the time specified in the `Sd_Basic_Acl_Timeout` option has passed, at which time the ACL files are reloaded.

Editing the Users File

The `users.cfg` file lists the username and password of each user authorized to connect to RTserver. The users list has this syntax:

```
users password_type {
    username password
}
```

where:

password_type represents how the password is stored. The only available storage is `plain`, meaning the password is stored in plain text.

username identifies an individual user. Each username is restricted to 64 characters.

password is the password used to authenticate *username*. The password size is unlimited. To specify no password, use empty quotation marks (`""`).

There are two default users in the `users.cfg` file, `admin` and `anonymous`. `admin` uses the password `smartsockets`, while `anonymous` does not require a password. For example:

```
users plain {
    admin smartsockets
    anonymous ""
}
```

To add new users, add the username and password to the list:

```
users plain {
    jdoe txY3s3
    mfrank foobar
    admin smartsockets
    anonymous ""
}
```

Editing the Groups File

The `groups.cfg` file defines groups of users that can share permission values. Group definitions simplify maintenance of permissions. For example, you might create a group for all developers, `dev`, knowing that all developers require the same set of permissions.

The groups list has this syntax:

```
group name {
    username
}
```

where:

name is the name of the group.

username identifies a user as a member of the group. The *username* must be listed in the `users.cfg` file before it can be added to a group.

There is one predefined group, the `admin` group:

```
group admin {
    admin
}
```

The only member of the `admin` group is the user `admin`.

You can have more than one group. For example:

```
group dev {
    jdoe
    mfrank
    anonymous
}

group admin {
    jdoe
}
```

In this example, there is only one member of the `admin` group, `jdoe`. The `dev` group includes two users plus the `anonymous` user.

The Admin Group

The `admin` group is reserved for users with administrative privileges, and prevents unauthorized users from publishing messages with the message type `CONTROL`. Only users who are included in the `admin` group can publish `CONTROL` messages. `CONTROL` messages are commonly used to send commands to other processes.

By default, the only user in the `admin` group is `admin`. Add or remove users from the group by editing the `groups.cfg` file. You can further define the `admin` group permissions in the `acl.cfg` file. You can restrict access to other message types with the option `Sd_Basic_Admin_Msg_Types`. See `Sd_Basic_Admin_Msg_Types` on page 555 for more information.

Editing the Permissions File

The `acl.cfg` file sets the permissions for each user or group.



Permissions are read from the bottom up. The first permission that applies to a user when the file is read is enforced.

All permissions use this syntax:

permission allow|deny user|group name host resource

where:

permission is the type of permission set. Valid settings are:

- `server` — establishes which RTservers are permitted to connect.
- `client` — defines which RTclients are permitted to connect connection.
- `membership` — establishes which user-groups, primarily multicast groups, an RTclient can join.
- `subscribe` — defines which subjects an RTclient can subscribe to.
- `publish` — defines which subjects an RTclient can publish to.

allow|deny indicates whether the permission setting grants or denies authorization.

user|group designates whether the permission applies to a user or a group.

name is the user or group whose permission is being defined. An asterisk applies to all users or groups.

host is the TCP/IP address of the host being granted access. *host* must be a numeric address in a series of numbers separated by periods (.). For example, 10.105.42.4. To indicate all hosts, use an asterisk (*). Use a partial address, such as 10.105. to indicate all hosts whose address begins with 10.105.

resource is specific to the permission being set:

- when *permission* is `server`, no resource is defined; you must use an asterisk.
- when *permission* is `client`, use the name of the SmartSockets project. An asterisk indicates all projects.
- when *permission* is `membership`, use the user-group name. An asterisk indicates all user-groups.
- when *permission* is `subscribe`, use the subject being subscribed to. To indicate all subjects, use ... or /...
- when *permission* is `publish`, use the subject being published to. To indicate all subjects, use ... or /...

The order of permissions is very important. Permissions are read from the bottom up. The first permission that applies to a user when the file is read is enforced. This is an example of a client connection permission:

```
client allow user *      * rtworks
client deny  user jdoe * rtworks
```

The top line allows all users to connect to the `rtworks` project. The bottom line prevents user `jdoe` from accessing that project. Notice that user `jdoe` matches both permissions. However, because the permissions are read from the bottom up, the last permission is enforced. `jdoe` is not permitted to connect to the `rtworks` project.



You must authorize the RTclients' unique subject, because the unique subject is not automatically authorized. If an RTclient is not authorized to receive messages sent to its unique subject, it will not receive administrative messages sent by RTserver. To allow all subscriptions to all RTclients' default unique subjects you can use the permission:

```
subscribe allow user * * /_*_*
```

Using Wildcards

Wildcards (* or ...) can be used in permissions files to indicate multiple users or groups, connections, hosts, and subjects. A wildcard component using an asterisk (*) never matches more than one component. Use ellipsis (...) or /... to indicate multiple levels of components.

For example, if subscribe permissions are granted on subject /dev/*, the client has permission to subscribe to /dev/misc but does not have permission to subscribe to /dev or /dev/misc/src. To specify all subjects use /... If the subject is not absolute, the RTserver's default subject prefix is used to prefix the subject.

Example 24 Example Permission File

This is an example `acl.cfg` file:

```
/* Server Permissions */
/* */
/* 1: allow all RTservers from all hosts to connect */
server allow user * * *

/* Client Permissions */
/* */
/* 1: allow all users in group admin when connecting from */
/* TCP/IP addresses beginning with 10.105. to any project */
/* 2: allow all users in group dev from all hosts to any project */
client allow group admin 10.105. *
client allow group dev * *

/* Subscription Permissions */
/* */
/* 1: allow all users from all hosts to subscribe to all subjects */
/* 2: deny all users from subscribing to /admin/... */
/* 3: allow users in group admin to subscribe to /admin/... */
subscribe allow user * * /...
subscribe deny user * * /admin/...
subscribe allow group admin * /admin/...

/* Publish Permissions */
/* */
/* 1: allow users in group dev to publish to /dev/... */
/* 2: deny user jdoe from from publishing to /dev/... */
publish allow group admin * /admin/...
publish deny user jdoe * /admin/...
```

Setting the Username and Password

Connecting RT processes must know the username and password assigned to them in the RTserver's `users.cfg` file. The username and password are set differently in daemon processes than they are in RTclients.

With RTclients, the username and password are set with the `TipcSrvSetUsernamePassword` function. With daemon processes such as RTserver and RTmon, you set the username and password by using two new command line arguments, `-username` and `-password`.

This example starts the RTserver and assigns the username `server` and password `foobar`:

```
rtserver -username server -password foobar
```



On platforms that support both 32- and 64-bit, use the `rtserver64` command to run the 64-bit version of RTserver.

An alternative to the command line argument and the `TipcSrvSetUsernamePassword` function is to use file-based credentials and the `Auth_Data_File` option. Set this option to a file containing the credentials you wish to send. File-based credentials are created with the RTacl tool.

For more information on the `TipcSrvSetUsernamePassword` function, see the *TIBCO SmartSockets Application Programming Interface*.

The RTacl Process

RTacl is a utility function provided as a debugging aid for users with complex access control list (ACL) permission files. RTacl is used to:

- evaluate ACL configuration files
- evaluate user permissions
- create credential files, when used with the `Auth_Data_File` option

RTacl has an interactive interface. To start RTacl, type at the command line:

```
$ rtacl
```

RTacl displays the `ACL>` prompt, at which you can execute RTacl commands. To exit from RTacl and return to the operating system prompt, enter `quit` at the RTacl command prompt. For a full list of commands supported by RTacl, see RTacl Commands on page 589.

RTacl must load the ACL files before running any commands that require an ACL, such as `evaluate`, `groups`, or `permissions`. Use the `load` command to specify the ACL configuration and verify its syntax. For example:

```
ACL> load $RTHOME/acl
```

Once the ACL is loaded, use RTacl to evaluate the ACL files. For example, you can use RTacl to discover:

- Whether a user has permission to subscribe to a particular subject (`evaluate` command).
- Which users are included in the ACL (`users` command).
- What groups are set and which users belong to each group (`groups` command).

Starting and Stopping RTserver

The `rtserver` command, described in detail in the *TIBCO SmartSockets Utilities* reference, is used to start and stop an RTserver. To start an RTserver, you can simply enter:

```
rtserver
```

at the operating system prompt. An optimized version of RTserver is started.



On platforms that support both 32- and 64-bit, use the `rtserver64` command to run the 64-bit version of RTserver.

The command must be entered with no filename extensions. Do not use `rtserver.x` or `rtserver.exe`. Using these extensions will prevent RTserver from running in the background and may cause other problems.

To start and use an RTserver, you must have a license for that RTserver. The license information must appear in the TIBCO license file or else the RTserver must be branded. For more information, see the *TIBCO SmartSockets Installation Guide*.

This is the syntax for the `rtserver` and `rtserver64` commands:

```
rtserver arg_list
```

where *arg_list* is optional and can consist of one or more of these command arguments, separated by a space:

- `-check` starts the non-optimized version of RTserver, which performs additional validations and checking. If you do not specify `-check`, by default the optimized version of RTserver is started. The optimized version is faster because validation and checking is turned off. However, without the validation and checking, it is harder to debug any problems. When developing your applications or testing RTserver in your development or production environments, always start RTserver with the `-check` argument.
- `-command filename` causes RTserver to look for a file named *filename* in the current directory (the directory from which RTserver is being run) and use that as a startup command file. Values for options specified in that file override values for the same options specified in the system-level or user-level startup command file.
- `-help` causes RTserver to print a brief synopsis and exit.

<code>-install -demandstart</code> <code>-autostart</code>	installs the RTserver process as a Windows service. You can specify either <code>-demandstart</code> or <code>-autostart</code> as the startup mode of the service. This option is supported only on Windows systems.
<code>-license</code>	displays license information about your RTserver. This is useful for finding out your license number when you need to contact TIBCO Product Support. Or you can display license information by using the <code>rtlic</code> shell script, described in the <i>TIBCO SmartSockets Utilities</i> reference.
<code>-no_console</code>	specifies not to display a Windows console associated with the detached process. This argument is ignored if you specify <code>-no_daemon</code> .
<code>-no_daemon</code>	specifies not to start RTserver as a background process and instead run in the foreground.
<code>-node node_name</code>	starts RTserver on a remote node. A remote shell command is used with <code>rsh</code> (<code>remsh</code> on HP-UX) to start RTserver on the remote node.
<code>-password pword</code>	provides the password used by RTserver, along with a username, to connect to other RTservers when Basic Security is enabled. Use with the <code>-username</code> argument. <i>pword</i> size is unlimited. To specify no password, use empty quotation marks ("")
<code>-server_names</code> <i>names_list</i>	specifies the value or values to use for the option <code>Server_Names</code> . <i>names_list</i> is a list of logical connection names separated by commas. It allows you to override the value or values specified in a startup command file. The values you specify here must use the same syntax as any value set for the <code>Server_Names</code> option.
<code>-stop</code>	specifies that a single RTserver should be stopped. The RTclients that are connected to that RTserver continue to run, eventually detect that RTserver has stopped, and try to find or start a new RTserver.
<code>-stop_all</code>	specifies that one RTserver, all RTservers connected to that RTserver, and all RTclients connected to all of the above RTservers should all be stopped. The RTclients are stopped by sending them a <code>CONTROL</code> message with a destination subject of <code>_all</code> and containing the command <code>quit force</code> .

<code>-stop_clients</code>	specifies that one RTserver and all RTclients connected to that RTserver should be stopped. The RTclients are stopped by sending them a CONTROL message with a destination subject of <code>_all</code> and containing the command <code>quit force</code> .
<code>-stop_servers</code>	specifies that one RTserver as well as all RTservers connected to that RTserver should be stopped. The RTclients that are connected to those RTservers continue to run, eventually detect that RTserver has stopped, and try to find or start a new RTserver.
<code>-threads <i>n</i></code>	<p>specifies whether the RTserver should start in MP multithread mode, and the number of threads to use. If you do not specify this argument, the default is 1 and RTserver starts in normal mode. If you specify a value greater than 1, the process checks to see if this RTserver was licensed for the MP option. If yes, the RTserver is started in multi-thread mode with the number of threads you specified for <i>n</i>. If no, you receive a warning message and the RTserver is started in single-thread mode.</p> <p>We do not recommend setting <i>n</i> to a large number. For more information, see <code>Server_Num_Threads</code>, page 569.</p> <p>The value you specify for <code>-threads</code> overrides the value specified for the <code>Server_Num_Threads</code> option in any of the RTserver startup command files.</p>
<code>-trace_file <i>filename</i></code>	specifies the name of the file where the RTserver puts debug information. The default, if you do not specify this argument, is the standard output (<code>stdout</code>) which is printed to the console.

<code>-trace_level</code> <i>level</i>	<p>specifies the amount of information the RTserver puts in the debug file (the file you specified in <code>-trace_file</code>). These are the values you can specify for level:</p> <ul style="list-style-type: none"> • <code>never</code>, no information is put in a debug file • <code>error</code>, only error messages put in debug file • <code>warning</code>, error and warning messages are put in debug file. This is the default setting. • <code>info</code> • <code>info_1</code>, provides thread information for multi-threading • <code>info_2</code> • <code>verbose</code> • <code>verbose_1</code> • <code>verbose_2</code> • <code>debug</code>, provides the maximum amount of information
<code>-uninstall</code>	removes the RTserver as a Windows service. This option is only supported on Windows systems.
<code>-username</code> <i>name</i>	<p>provides the username used by RTserver, along with a password, to connect to other RTservers when Basic Security is enabled. Use with the <code>-password</code> argument.</p> <p><i>name</i> size is restricted to 64 characters.</p>
<code>-verbose</code>	is the same as specifying <code>-trace_level info</code> . The <code>-verbose</code> argument is provided for backwards compatibility. The preferred method for specifying the amount of information for RTserver to provide is to use the <code>-trace_level</code> argument.
<code>-version</code>	causes RTserver to print version and revision levels.

Notes:

- All the arguments for the `rtserver` command are optional.
- When you specify any of the `-stop` arguments, any other arguments, except `-server_names`, in the command are ignored.
- Previous releases supported the `-cmd_mode` argument, which started the RTserver in interactive command mode, but this mode is no longer supported, and so the `-cmd_mode` argument has been eliminated.

Starting RTserver

RTserver runs as a background process (on OpenVMS and Windows this is known as a detached process, on MVS RTserver can run as a started task) without an interactive command interface. RTserver can be started manually from the operating system prompt, or it can be started automatically when an RTclient first tries to connect to RTserver. If you want RTclient to be able to start RTserver automatically, you must use one of the non-default start prefixes. See Start Prefix on page 195 for more on start prefixes.

On Windows, if RTserver was installed as a Windows service, the proper environment variable must be set before an RTclient can automatically start an RTserver when it attempts to connect. Set the RTSERVER_CMD environment variable to:

```
net start "SmartSockets RTserver"
```



Automatic starting of 64-bit RTserver on Windows is not supported at this time.

For details on invoking RTserver, see the `rtserver` reference information in the *TIBCO SmartSockets Utilities* reference.

Here are the steps to start RTserver:

1. Change to the directory in which RTserver will run.
2. Create or edit startup command files, if applicable (see Startup Command Files, page 498).
3. Invoke the RTserver executable.

The first thing you must do to begin running RTserver is to change from the current working directory to the one that contains the RTserver command file, `rtserver.cm`. If no `rtserver.cm` file is needed, then RTserver can be started from any directory.

To change directories, use:

UNIX:

```
$ cd directory
```

OpenVMS:

```
$ SET DEFAULT [directory]
```

Windows:

```
$ cd directory
```

To start RTserver on UNIX systems, type this command at the operating system prompt:

```
$ rtserver
```

You should see the SmartSockets banner information and a line saying that the RTserver was started successfully.

To start RTserver on Windows systems, go to the Start menu, select Programs, and select the SmartSockets program folder. Select RTserver. Or, type this command at the SmartSockets command prompt:

```
$ rtserver
```

If RTserver started successfully, the window is automatically minimized. Check your bottom bar for the RTserver process and click on it to expand the window. In the window, you should see a message saying that the RTserver was started successfully.



On platforms that support both 32- and 64-bit, use the `rtserver64` command to run the 64-bit version of RTserver.

Stopping RTserver

To stop a running RTserver process, use the `rtserver` command with the `-stop` argument on the computer that RTserver is running on (add the command-line arguments `-server_names node` to stop RTserver on a remote node). This mode connects to a running RTserver and requests that RTserver gracefully shut down. It should normally never be necessary to stop and restart RTserver.

Here is an example:

```
$ rtserver -stop
```

When RTserver receives a stop request, it first uses the value of the option `Enable_Stop_Msgs` to check if stopping RTserver is enabled, and if allowed, RTserver then exits. Note that the `rtserver -stop` command shuts down RTserver even if there is a project running. To stop RTserver and all RTclients that are connected to that RTserver, use `rtserver` with the `-stop_clients` argument. To stop RTserver and all RTserver processes that are connected to that RTserver, use `rtserver` with the `-stop_servers` argument. To stop RTserver, all RTclients that are connected to that RTserver, and all RTserver processes that are connected to that RTserver, use `rtserver` with the `-stop_all` argument. See `Enable_Stop_Msgs`, page 534 for details on how to properly configure RTserver shutdown security.

Working with RTserver

The following sections discuss how RTserver works and how to configure RTserver. RTserver does not have any API functions that you can use, but it provides many options and commands for customizing the behavior of RTserver.

Setting Options

The primary way of configuring RTserver is through options in startup command files. This is especially important for RTserver, as RTserver can be automatically started by RTclient. If RTserver is not configured properly, it may not run correctly or even start at all. All RTserver startup command files and options are discussed in detail in Chapter 8, Options Reference.

Creating Connections

RTserver creates two distinct groups of connections. The first group of connections is specified in the option `Conn_Names`. These connections are server connections that are used by other processes (both RTclient and RTserver) to find this RTserver. The second group of connections is specified in the option `Server_Names`. These connections are client connections that are used to find other RTservers.

The maximum number of RTclients allowed to connect to an RTserver is specified by the RTserver option `Max_Client_Conns`. This option is useful for limiting the amount of publish-subscribe connection resources that each RTserver must provide.

Logical Connection Names

An RTserver uses logical connection names much like an RTclient. Because the usage is similar, only the differences in function are discussed in this section.

RTserver uses logical connection names in the options `Conn_Names` and `Server_Names`. RTserver uses the option `Default_Protocols` differently for `Conn_Names`. If a protocol is not listed in a name, then RTserver traverses the protocols in `Default_Protocols` and creates a server connection using each protocol. RTserver does not use any of the logical connection name modifiers of the type start prefix in either `Conn_Names` or `Server_Names`. RTserver never starts any RTclients or RTservers. RTserver supports `_random`, just like RTclient.

Logical Connection Name Modifiers

Each logical connection name in RTserver may be prefixed with one or more modifiers. The modifiers can be in any order and must be separated by a colon. For example:

modifier: modifier: protocol: node: address

The modifier can be a keyword or a name-value pair, as in *keyword:name=value:protocol:node:address*. There are several logical connection name modifiers supported by RTserver including: connect modifier and connection cost modifier. RTclient supports the start prefix modifier.

Controlling How RTserver Connects to Other RTservers

Each logical connection name in RTserver can also have a connect modifier prefixed. A connect modifier is a keyword modifier. For example:

connect_modifier: protocol: node: address

The connect modifier controls how an RTserver process tries to connect to other RTservers. When an RTserver connects to another RTserver process listed in the `Server_Names` option, the other RTserver responds with a list of other RTservers that the first RTserver may also connect to. The valid keywords for a connect modifier are:

<code>connect_all</code>	also connect to all RTservers to which the other RTserver is connected
<code>connect_one</code>	connect only to the other RTserver, and not to all the RTservers to which it is connected
<code>connect_all_stop</code>	also connect to all RTservers to which the other RTserver is connected, and stop traversing <code>Server_Names</code> if the first connection succeeds
<code>connect_one_stop</code>	connect only to the other RTserver, not to all the RTservers to which it is connected, and stop traversing <code>Server_Names</code> if the first connection succeeds

If no connect modifier is specified in a logical connection name in the `Server_Names` option in RTserver, the connect modifier from the option `Default_Connect_Prefix` is used. The default value for `Default_Connect_Prefix` is `connect_one`.

Assigning a Cost to RTserver to RTserver Connections

Each logical connection name in RTserver can also have a cost modifier prefixed. A cost modifier is a name-value modifier.

For the purposes of dynamic message routing, each RTserver to RTserver connection is given a cost of one by default. This can be thought of as one hop between the RTservers at each end of the connection. However, the physical connections between two RTservers can vary greatly in terms of bandwidth, load, reliability and other attributes. A logical connection name listed in `Server_Names` can assign a cost to the physical connection that it represents. A cost of one is the default and is the least expensive cost that can be assigned to a connection. A cost greater than one will be factored into the lowest cost path message routing algorithm and will result in more message traffic being routed through other, less expensive, available connections. A cost is assigned to a connection by preceding a logical connection name with a cost modifier of the form `cost=n`, where `n` is a positive integer, separated by a colon:

cost=n:protocol:node_address

For example, suppose the option `Server_Names` for RTserver on node `shemp` is set to:

```
cost=5:tcp:larry:8001,cost=2:tcp:curly:8001,cost=10:tcp:moe:8001
```

If the RTservers are running on `larry`, `curly` and `moe` and do not attempt to connect to `shemp` then the RTserver on `shemp` would establish connections to RTservers on `tcp` port 8001 on the nodes `larry`, `curly` and `moe`. The RTserver connection between `shemp` and `larry` would be assigned a cost of five, the connection between `shemp` and `curly` a cost of two and the connection between `shemp` and `moe` a cost of ten.

If both RTservers on the ends of a connection try to assign a different cost to the connection, the results will be indeterminate. While traversing the `Server_Names` list, each RTserver will attempt to create the connection by making a connect request. One RTserver should connect successfully. The other RTserver should issue a successful accept. The second RTserver's connect attempt will fail because the connection will already exist at that point. The RTserver whose connect succeeds will assign the cost to the connection. The cost will be one or the other cost, but it is indeterminate which one. So the RTserver that successfully initiates the connection has precedence if the two RTservers on each end of the connection try to assign different costs to the connection. See *Lowest Cost Message Routing* on page 301 for more information.

Finding Other RTserver Processes

Because RTserver never starts any other RTclients or RTservers, it does not need any of the `Server_Start_*` options to control how it finds other RTservers. RTserver traverses the list of logical connection names in `Server_Names` once and uses each name to attempt to create a client connection to another RTserver process. If RTserver can connect to another RTserver process, it joins the existing group of multiple RTservers (see Multiple RTserver Processes on page 298 for more details on this group). The RTserver `connect` and `disconnect` commands can be used to dynamically join and leave the group of multiple RTservers, and create and destroy connections to other RTservers.

Reconnecting to Other RTserver Processes

If an RTserver loses its connection or cannot connect to another RTserver process listed in the `Server_Names` option, it will automatically try to reconnect at a regular interval specified by the `Server_Reconnect_Interval`. Only the RTserver that makes the initial connection attempt will try the reconnect, otherwise temporary deadlock could occur if both RTservers simultaneously try to connect to each other. The automatic reconnection feature can be disabled by setting the option `Server_Reconnect_Interval` to `0.0`.

Receiving and Processing Messages from RTclient

For most messages that RTserver receives from RTclient, RTserver looks at the destination property (a subject name) of the message and routes the message on to the RTclients subscribing to that subject. A few message types are processed locally, though. Table 13 lists the message types for which RTserver creates a process callback on the connections to RTclients.

Table 13 Message Types that RTserver Processes from RTclient

Message Type	Description
CONTROL	RTserver executes command if the message destination is <code>_server</code> , otherwise RTserver routes the message.
SUBJECT_SET_SUBSCRIBE	RTclient wants to start or stop subscribing to a subject.
DISCONNECT	current value of <code>Server_Disconnect_Mode</code> option; also provides for a more graceful disconnect (RTserver does not have to detect end-of-file (EOF)).

Table 13 Message Types that RTserver Processes from RTclient (Cont'd)

Message Type	Description
GMD_ACK	GMD acknowledgment of successful delivery
GMD_DELETE	request to terminate GMD for a specific message
GMD_INIT_CALL	request to initialize GMD or load balancing accounting in RTserver for a subject to which messages will be published
GMD_STATUS_CALL	request for current GMD status of a specific message
MON_*_SET_WATCH (11 message types)	see Chapter 5, Project Monitoring
MON_*_POLL_CALL (21 message types)	see Chapter 5, Project Monitoring
SERVER_STOP_CALL	RTclient wants to stop RTserver

Processing CONTROL Messages From RTclient

When RTserver receives a CONTROL message from RTclient, RTserver looks at the destination property (a subject name) of the message and routes the message on to the RTclients subscribing to that subject unless the destination is the unique subject of some RTserver or `_server`. If the destination is the unique subject of an RTserver, then the message is routed to that RTserver. If the destination is `_server`, RTserver uses the value of the option `Enable_Control_Msgs` to check if the command is enabled, and if allowed, then RTserver executes the command in the CONTROL message by calling the function `TutCommandParseStr`. This method allows RTserver to both route CONTROL messages and receive remote commands from RTclient. CONTROL and ADMIN_SET messages are the only message types that RTclient can explicitly publish to an RTserver. See `Enable_Control_Msgs`, page 533 for details on how to properly configure CONTROL message security.

Message File Logging

As described in Message File Logging Categories on page 211, RTclient can log three categories of messages (data, status, and internal) to message files for incoming and outgoing messages. RTserver has a similar capability, but the message types are divided into only two categories:

- client — messages sent to and received from RTclients
- server — messages sent to and received from other RTservers

Both standard and user-defined message types are logged, but user-defined message types are always logged using the verbose format (see Grammar on page 32 for more information on the verbose message file format).

Logging Messages

RTserver starts and stops logging messages in the logging categories by setting these options:

To Log These Messages:	Use this Option:
Incoming from clients	Log_In_Client
Outgoing to clients	Log_Out_Client
Incoming from servers	Log_In_Server
Outgoing to servers	Log_Out_Server

Dynamic Message Routing

Because of the high performance publish-subscribe message routing features of RTserver, a single RTserver often is sufficiently powerful for small workgroup-sized projects. Large-scale distributed systems for the intranet, extranet, and Internet, however, need the benefit of enterprise-wide publish-subscribe capabilities. Dynamic message routing offers the scalability and flexibility these large systems require.

Dynamic message routing has these features.

- multiple RTservers can be grouped in any arbitrary connection topology with or without redundant routes
- optimized lowest cost routing of messages including same-first hop optimizations
- connections within the topology can be configured with different costs for improved message routing
- automatic rerouting of messages upon topology changes (that is, whenever an RTserver connects or disconnects)
- a distributed publish-subscribe database model where each RTserver only knows what it needs to know to operate successfully
- RTserver itself can subscribe to subjects to provide more efficient subject routing and reduce network chatter

Why is Dynamic Message Routing Needed?

When in a WAN environment such as the Internet, TCP/IP has many sophisticated, evolving technologies to ensure reliable delivery of network packets. Internet standards such as Domain Naming Service (DNS) and Open Shortest Path First (OSPF) routing all help TCP/IP scale to the levels it has reached today. As described in *Why is GMD Needed?*, page 312, though, there are gaps in this reliability that any large distributed system must overcome. In addition, there are areas such as publish-subscribe and real-time directory services updating that TCP/IP simply does not address (other products such as DCE and X.500 do not address real-time updates, either).

It might seem redundant for SmartSockets publish-subscribe to offer dynamic message routing when TCP/IP has many seemingly similar dynamic routing features at the network packet level. The global TCP/IP-based Internet is inherently a peer-to-peer system, such as the Web, FTP, email, or Telnet client connecting to the Web, FTP, email, or Telnet server on a specific machine, and does not have the flexible one-to-many publish-subscribe capabilities required for large-scale distributed systems.

Dynamic message routing takes full advantage of TCP/IP's dynamic routing capabilities and extends them with the power of publish-subscribe. Dynamic message routing is also needed for mixed protocol networks where not every computer supports TCP/IP. TCP/IP can also be used as the backbone, and yet dynamic message routing can deliver messages off to the interconnected networks running Netware or any other protocol.

Multiple RTserver Processes

In addition to routing messages among RTclients, multiple RTservers also dynamically route messages to each other. Multiple RTservers can distribute the load of publish-subscribe message routing. If a project is partitioned so that most of the messages being sent are routed among processes on the same node, then the use of multiple RTservers can reduce the consumption of network bandwidth (processes on the same node can use the non-network local IPC protocol).

The RTclients that are directly connected to an RTserver are called direct RTclients, and, likewise, the RTclients directly connected to other RTservers are called indirect RTclients. Similar terminology is used for direct RTservers and indirect RTservers.

Figure 25 Process Connectivity With RTserver Cloud

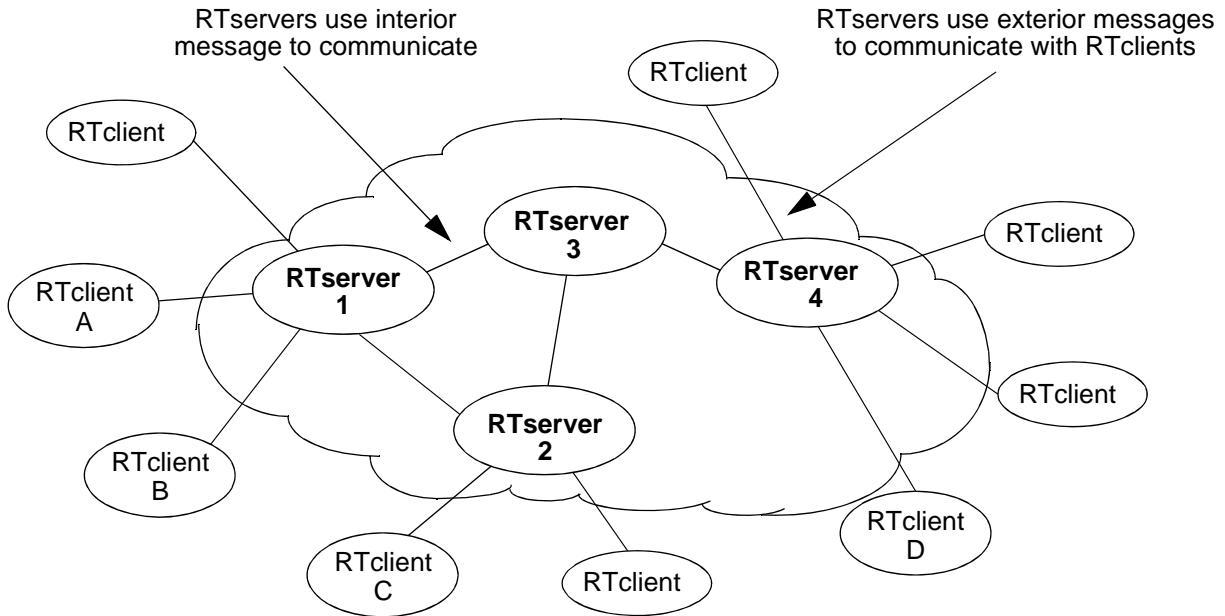


Figure Notes:

- RTclients A and B are direct RTclients of RTserver 1.
- RTclients C and D are indirect RTclients of RTserver 1.
- RTservers 2 and 3 are direct RTservers of RTserver 1.
- RTserver 4 is an indirect RTserver of RTserver 1.
- RTservers 1, 2, 3, and 4 form a group. They do not form a matrix.

Multiple RTservers can form a group that allows SmartSockets projects great flexibility, scalability, and robustness. A group is an arbitrary topology of interconnected RTservers, which is represented with an undirected graph and uses graph algorithms to calculate lowest cost path between two RTservers. Figure 25 shows an example of a group with four RTservers. There can be more than one RTserver group on a network, but each group is self-contained. Just as RTclients in different projects cannot publish messages to each other, RTservers in different groups cannot send messages to each other.

Figure 26 RTserver Groups Connected Using Gateways over a WAN

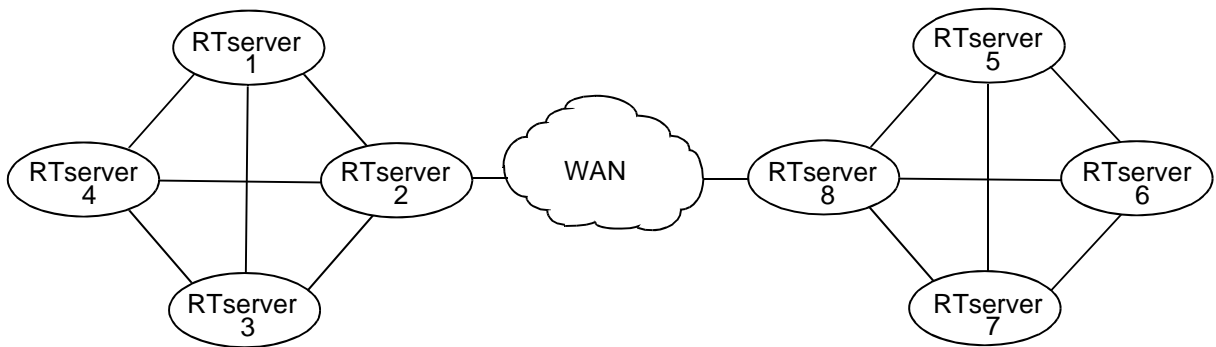


Figure 26 shows a typical RTserver group where some of the communication must occur over a WAN. Rather than have all the RTservers interconnected, RTserver 2 and RTserver 8 act as gateways to the WAN and limit the number of connections occurring over the WAN.

The group approach decreases the number of connections (and thus operating system resources such as socket file descriptors) each RTserver needs, but publish-subscribe message routing for SmartSockets becomes more complex. However, the message routing is handled completely transparently and in real time. The RTservers in the group send messages among themselves to update each other regarding subscribes and unsubscribes. When RTclient connected to RTserver subscribes to a subject, that RTserver is also conceptually subscribing to that subject from the other RTservers. When no RTclients connected to RTserver are subscribing to a subject, then that RTserver does not receive any messages for that subject from other RTservers in the group (this can be overridden for greater efficiency as described in RTserver Subscribes on page 303).

The speed at which a message is routed and delivered depends on many factors including how many connections the message has to pass through and also how fast those connections are. For example, local connections on UNIX and MVS are always much faster than TCP/IP network connections. Each RTserver manages the majority of publish-subscribe message routing for its direct RTclients. It is transparent to RTclients how many RTservers are in the group. From the

standpoint of RTclient, the group is a virtual RTserver cloud that happens to span several nodes. RTclient simply needs to connect to a single RTserver in the cloud and can then send a message to or receive a message from any RTclients connected to the cloud.

Distributed Publish-Subscribe Database

In a large distributed publish-subscribe system, if every RTserver knows about all the RTclients and all subject subscriptions, the system experiences scalability problems due to network chatter that occurs each time the state of the system changes (subscription and RTclient changes). In SmartSockets, RTserver does not have complete knowledge. Instead the RTserver group can be viewed as a distributed real-time database or directory service of publish-subscribe information. The limited information maintained by each RTserver makes SmartSockets publish-subscribe work easily and efficiently, and scale to very large projects. Because the information is distributed, there is no single point of failure in the system.

Just as RTserver only routes messages to RTclients that are subscribed to a subject, RTserver only routes messages to other RTservers that have at least one direct RTclient subscribing to a subject. In general, when RTserver receives a message from another RTserver, it routes the message to its direct RTclients and sometimes also to other specific RTservers. When RTserver receives a message from a direct RTclient, it might route that message to both direct RTclients and other direct RTservers.

Lowest Cost Message Routing

The RTserver group uses interior messages to communicate and synchronize with each other. The exterior messages are those routed for RTclients (normal publish-subscribe). The monitoring messages described in Chapter 5, Project Monitoring, have characteristics of both interior and exterior messages. These interior messages use a form of memory-only GMD, complete with acknowledgments and resends, to ensure delivery and ordering so as to prevent corruption of the RTserver distributed database. When there is a topology change in the group of RTservers (due to a new connection being created or an existing connection being destroyed), all appropriate GMD messages are resent in sequence number order to prevent out-of-order delivery and message loss.

RTserver uses lowest cost routing, where each RTserver only knows the first-hop to a destination RTserver, which simplifies accounting. Dijkstra's shortest-path algorithm is used. Each RTserver runs the same-first hop calculation in parallel. It follows that each RTserver does know about all RTservers in the group.

Figure 27 RTserver Cloud with Default Connection Costs

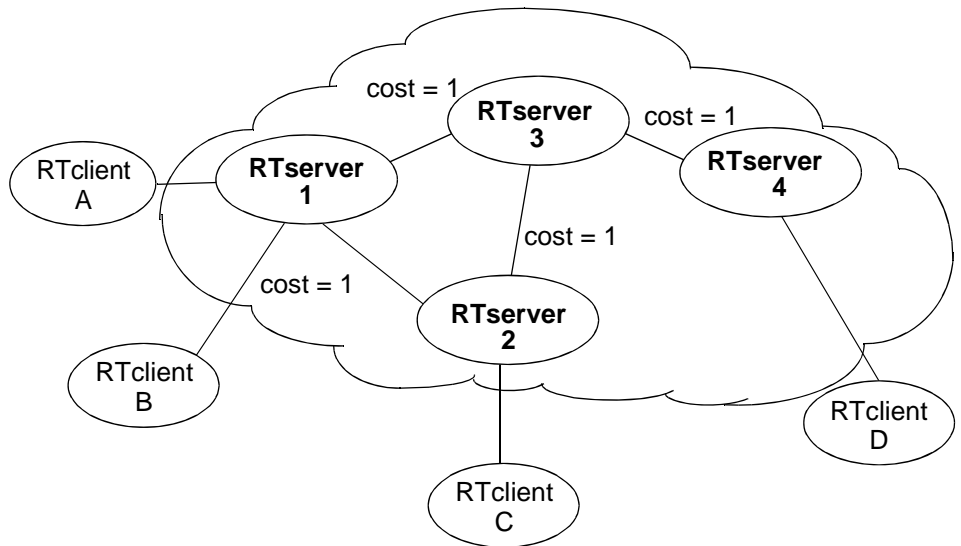


Figure 27 shows a SmartSockets application with multiple RTservers and RTclients. All RTserver-to-RTserver connections have the default connection cost of one. In Figure 27 a message going from RTclient A to RTclient B would go through one RTserver. A message going from RTclient A to RTclient C travels through two RTservers, RTserver1 and RTserver2. A message going from RTclient A to RTclient D travels through three RTservers, namely RTserver1, RTserver3 and RTserver4.

Routing is dynamic in that it can change at any time. Whenever a new RTServer becomes available or an existing RTServer goes down, routing tables in the RTServers are updated in real time to reflect the new topology. Lowest cost routing can be made even more efficient by changing the relative costs of RTServer to RTServer connections to closely match available bandwidth, load conditions, and so on, and by using RTServer subscribes as shown in the next section.

Figure 28 RTServer Cloud with Non-Default Connection Costs

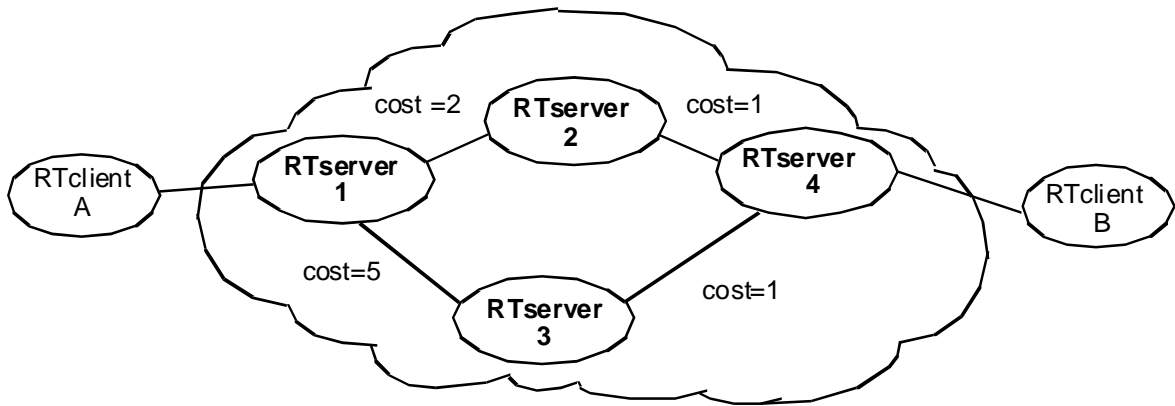


Figure 28 shows a SmartSockets application with multiple RTServers and RTclients. Two RTServer-to-RTServer connections have non-default connection costs. In Figure 28, there are two possible routes for a message going from RTclient A to RTclient B. The route starting at RTserver1 going through RTserver2 and finally through RTserver4 has a cost of $2+1=3$. The alternate route starting at RTserver1 going through RTserver3 and finally through RTserver4 has a cost of $5+1=6$. With this configuration, all messages going from RTclient A to RTclient B will be routed along the first route, going from RTserver1 to RTserver2 and on to RTserver 4 because that is the lowest cost route. If the connection between RTserver1 and RTserver2 becomes unavailable, then the messages going from RTclientA to RTclientB will be routed using the RTserver1 to RTserver3 to RTserver4 route.

When a message is routed from one RTServer to several other RTServers, only the minimum necessary copies of messages are sent between the RTServers. That is, if a message is sent to several other RTServers through the same first-hop connection, then the message is marked as needing to be delivered to several RTServers, and one copy, not several, is sent through the proper first hop. This is superior to a flooding algorithm when there are loops in the group topology and the same message could be sent over and over.

RTserver Subscribes

SmartSockets provides many mechanisms to allow projects to scale to large numbers of machines. One important technique is to minimize network chatter whenever possible. One method of doing this is through the use of RTserver subscribes, which are specified through the RTserver `subscribe` command.

This is useful where there is a network bandwidth problem or a usage problem, such as:

- very large projects where you have many RTclients or many subjects
- projects where some of the network connections are slow or expensive

The `subscribe` command can be executed from a command file, from the RTserver command line, or it can be sent to RTserver from another process through a CONTROL message. The general syntax of the `subscribe` command is:

```
subscribe project subject
```

For example, to subscribe to all subjects underneath `/stocks` in the `stock_trader` project, this `subscribe` command is used:

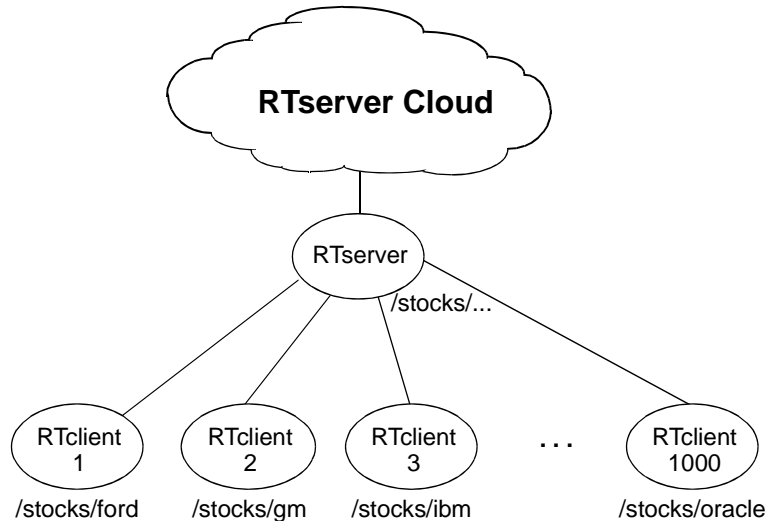
```
subscribe stock_trader /stocks/...
```

Additional subscribes are additive. For example, if you issue a command to subscribe to the subject `/stocks/ford`, and then issue a second command to subscribe to `/stocks/gm`, the RTserver then subscribes to both subjects. There is an associated `unsubscribe` command to have RTserver unsubscribe from a subject. See `subscribe`, page 626 for more details.

RTserver subscribes allow any number of RTclient subscribes to be combined into a single update. This can greatly reduce the amount of network traffic that occurs as RTclients connect and disconnect and as they subscribe and unsubscribe to subjects. The scalable power of SmartSockets publish-subscribe is realized by the power of the hierarchical subject namespace combined with the RTserver subscribes to efficiently cover hundreds or thousands of RTclient subscribes.

For example, consider the simple environment shown in Figure 29. There are 1,000 RTclients connected to a single RTserver, each subscribed to a single subject, such as `/stocks/ford`.

Figure 29 Benefits of RTserver Subscribes



This adds up to 1,000 subscription messages for the RTserver to send to all other RTservers, which can cause a significant amount of network message traffic if there are many RTservers, or the RTclients are starting/stopping or subscribing/unsubscribing frequently. If this RTserver itself subscribes to `/stocks/...`, then the RTserver matches all of the 1,000 RTclient subscribes, and there is only one subscription message, not 1,000 being sent to all other RTservers.

It is important to note that the primary purpose of RTserver subscribes is to increase the efficiency of the distributed system. In all cases, SmartSockets publish-subscribe, GMD, and monitoring all work in any RTserver topology without any manual configuration. Using RTserver subscribes, it is possible that a message may travel all the way to the subscribed RTserver before being discarded if there is no direct RTclient subscribed to the subject.

Network Considerations

Controlling Network Bandwidth and Usage

It can be useful to regulate the amount of bandwidth used by server processes such as the RTserver and RTgms in your system. To regulate RTserver and RTgms bandwidth usage, a token bucket algorithm is used as a model. Tokens are accumulated steadily as time passes. The amount of data the RTserver or RTgms process is permitted to pass into the network for a particular connection is indicated by the number of tokens that have accumulated for that connection. Each token is a byte of data.

Bandwidth Rate Control Options

Several options have been added to implement this model. For each type of connection that an RTserver has (to an RTclient, to another RTserver, to an RTgms process), there is:

- an option that specifies the rate at which tokens accumulate for the connection
- an option that specifies the maximum number of tokens allowed to accumulate for the connection
- an option that specifies the amount of time RTserver waits, after it runs out of tokens, for more tokens to accumulate before resuming the sending of data

When you set these options for an RTserver, they apply to the data and messages RTserver sends across the specific type of connection. They do not apply to data or messages received by the RTserver.

These options also exist for RTgms, for the group channel between an RTgms process and an RTserver. When you set these options for RTgms, they apply only to the data and multicast messages RTgms sends up the group channel to an RTserver.

For more information, see the reference material on these options in Chapter 8, Options Reference:

- Client_Burst_Interval
- Client_Max_Tokens
- Client_Token_Rate
- Group_Burst_Interval
- Group_Max_Tokens
- Group_Token_Rate

- `Server_Burst_Interval`
- `Server_Max_Tokens`
- `Server-Token_Rate`

Setting Bandwidth Rate Control Options

For connections from the RTserver, these options are set in the RTserver startup command file (`rtserver.cm`) and apply to all connections of each type (RTclient, RTserver, RTgms). To set the options for an individual connection, you can set them dynamically with the `T_MT_ADMIN_SET_OUTBOUND_RATE_PARAMS` message.

For the group channel from an RTgms process to an RTserver, these options are set in the RTgms startup command file (`rtgms.cm`) and apply to all group connections. To set the options for an individual group connection, you can set them dynamically with the `T_MT_GRP_ADMIN_SET_OUTBOUND_RATE_PARAMS` message.

When you send an `ADMIN_SET` message to a particular RTserver or RTgms process, the options apply to outbound data sent on the specified connection.

For example, if there is a group channel between an RTserver and an RTgms process, using `T_MT_ADMIN_SET_OUTBOUND_RATE_PARAMS` and specifying a group name sets the options for RTserver for that group channel, controlling the data the RTserver sends to RTgms.

If you use `T_MT_GRP_ADMIN_SET_OUTBOUND_RATE_PARAMS` and specify the same group name, the options are set for RTgms, and control the data the RTgms sends to RTserver on that same group channel. To control the bandwidth in both directions for a group channel, you must set bandwidth rate control options for both the RTserver and the RTgms.

For a description of the `ADMIN_SET` message for RTserver, see RTserver Options, page 495 in Chapter 8, Options Reference. For a description of the `ADMIN_SET` message for RTgms, see RTgms Options, page 650 in Chapter 10, Using Multicast.

Handling Network Failures In Publish Subscribe

RTserver and RTclient can take full advantage of the features of connections that detect and recover from network failures. For an introduction to these features, see *Handling Network Failures In Publish Subscribe* on page 307.

Many of the RTserver and RTclient features already discussed in this chapter help to add fault tolerance to SmartSockets:

- RTclient can automatically start RTserver if one is not already running (see *Finding and Starting RTserver* on page 197). Note: to allow RTclient to automatically start RTserver, you must use one of the non-default start prefixes described in *Start Prefix* on page 195.
- RTclient can restart RTserver and reconnect if an error occurs (see *Automatically Reconnecting to RTserver* on page 201). There are also APF authorization considerations for automatic start and restart on MVS.
- RTclient can run with a warm connection to RTserver if the RTserver is temporarily unavailable, such as when a remote node crashes and needs several minutes to reboot (see *Warm Connection to RTserver* on page 235).
- RTclient can monitor projects, subjects, RTclients, and RTservers for problems. If a problem is detected, the RTclient makes the appropriate decision to correct the problem (see Chapter 5, *Project Monitoring*).
- RTclient and RTserver can use a list of logical connection names to search for other processes (see *Logical Connection Names for RT Processes* on page 192).
- RTclient and RTserver have many options, providing a large degree of control over how many times certain operations are attempted, such as `Server_Start_Max_Tries`, and how long of a period of time those operations have to complete, such as `Server_Start_Timeout`, (see *Creating a Connection to RTserver* on page 189).
- A backup RTclient can use the option `Server_Msg_Send` to receive incoming messages but not send any outgoing messages (see *Sending Messages* on page 208).
- RTserver can reconnect to other RTservers if an error occurs (see *Reconnecting to Other RTserver Processes* on page 293).
- Dynamic message routing handles many network failures, such as rerouting around RTservers that fail (see *Dynamic Message Routing* on page 296).
- Load balancing enables publish-subscribe to have redundant RTclients for processing messages (see *Load Balancing* on page 215).

In addition to the above features, both RTclient and RTserver have several additional options that can be easily configured to add fault tolerance.

RTclient Options

In addition to the already mentioned fault tolerance features, these convenient options can be used to help RTclient check for network failures:

- `Server_Keep_Alive_Timeout` — controls the keep alive timeout property of the connection to RTserver
- `Server_Read_Timeout` — controls the read timeout property of the connection to RTserver
- `Server_Write_Timeout` — controls the write timeout property of the connection to RTserver

RTserver Options

In addition to the already mentioned fault tolerance features, these options can be used to help RTserver check for network failures:

- `Client_Connect_Timeout` — controls how long RTserver waits when trying to read a `CONNECT_CALL` message from a new RTclient that has just connected
- `Client_Max_Buffer` — controls how many bytes of data RTserver buffers for an RTclient before deciding the RTclient has failed (this is needed instead of keep alives because RTclient is not required to process messages from RTserver at regular intervals)
- `Server_Connect_Timeout` — controls how long RTserver waits when trying to read a `SRV_CONNECT_CALL` message from a new RTserver process that has just connected
- `Server_Keep_Alive_Timeout` — controls the keep alive timeout property of the connections to other RTservers
- `Server_Read_Timeout` — controls the read timeout property of the connections to other RTservers
- `Server_Reconnect_Interval` — controls the rate at which RTserver tries to reconnect to other RTservers

Chapter 4

Guaranteed Message Delivery

Under normal operation in SmartSockets, all messages sent through connections are delivered successfully and processed in a timely manner. If a network failure occurs, though, data can be lost. For some applications, such as bank transactions or Internet commerce, missed messages or duplicate messages are unacceptable. The features described in *Handling Network Failures* on page 149 allow connections to detect network failures and initiate recovery. Guaranteed message delivery (GMD) fully recovers from these failures and ensures that messages are transmitted as needed.

The first part of this chapter describes the features of connections that implement GMD. The second part of this chapter describes the additional features you get when you use GMD with publish-subscribe. For a discussion of the GMD features specific to RTserver and RTclient that add more function, such as RTserver continuing to buffer messages for RTclient processes that have temporarily failed, see *Publish-Subscribe and GMD* on page 344.

Topics

- *Features of GMD, page 311*
- *Why is GMD Needed?, page 312*
- *File-Based and Memory-Based GMD, page 314*
- *GMD Composition, page 315*
- *Working With GMD, page 322*
- *Configuring GMD, page 331*
- *Sending Messages, page 336*
- *Receiving Messages, page 337*
- *Acknowledging Messages, page 338*
- *Resending Messages, page 339*
- *Handling GMD Failures, page 341*
- *Limitations of GMD, page 343*

- *Publish-Subscribe and GMD, page 344*
- *RTclient GMD Considerations, page 350*
- *RTserver GMD Considerations, page 354*
- *Combining GMD and Monitoring, page 356*
- *Handling GMD Failures with RTclients and RTservers, page 357*

Features of GMD

In general, GMD has these features:

- easy configuration with the options `Unique_Subject` and `IpC_Gmd_Directory`
- easy usage by setting the delivery mode property of a message with the functions `TipCMsgSetDeliveryMode` and `TipCMtSetDeliveryMode`, or delivery timeout property of a message with the functions `TipCMsgSetDeliveryTimeout` and `TipCMtSetDeliveryTimeout`
- persistence of messages in disk files in case a program crashes and is restarted
- transparent operation with automatic file management and acknowledgment of delivery
- notification when GMD fails, to allow flexible user-defined recovery procedures
- performance is limited only by performance of local file system and network

Why is GMD Needed?

As discussed in *Sockets* on page 85, connections use (stream) sockets to transmit messages. Sockets have several well-defined, useful features that connections build upon. For example, sockets buffer a fixed amount of data for better performance. The API functions for working with sockets provide an inherently synchronous model. For example, the receive operation by default blocks the receiving process until data is available. The operating system underneath the API is usually very asynchronous. For example, it has interrupt-driven methods to move the socket data from one buffer over the network to another buffer.

Loss of Data When Sockets Fail

When data is sent through a socket, the send operation by default returns as soon as the operating system buffers the data for future delivery over the physical network. The data that is delivered successfully is received in the order it was sent. If there is a network failure, which requires the socket to be closed, such as a node crashing, there is no way for the sending process to know how much, if any, of the data was lost! SmartSockets GMD builds upon the useful properties of sockets and overcomes the deficiencies caused by the limitations of sockets.

Acknowledgment of Delivery

Because of the internally asynchronous nature of stream sockets, GMD needs some way to determine when a message has been successfully delivered. At the socket level, reliable protocols like TCP/IP are built on top of unreliable physical networks where data can be lost, such as Ethernet or serial lines. These network protocols ensure an ordered byte stream by:

- sending their own acknowledgment packets to notify the sending operating system of successful reception by the receiving operating system
- resending lost data
- resequencing packets that arrive out of order

Sockets do not provide any capability to notify the sending process that the receiving process has successfully received the data and acted on it. Sockets are thus reliable except when a serious error occurs that requires the socket to be closed. It is this situation where a network programmer most needs to know what data was lost.

GMD also uses acknowledgments in the form of acknowledgment messages. The connection-level acknowledgment messages are different from the socket-level acknowledgment packets, but serve the same purpose: the connection-level function notifies the connection, and the socket-level function is for the operating system. From this point on, the term acknowledgment is used to refer to a SmartSockets-level acknowledgment message, not a socket-level acknowledgment packet.

Alternatives to Stream Sockets

Using stream sockets for any IPC development has both advantages and disadvantages, but in general the advantages greatly outweigh the disadvantages. In TCP/IP networks, the alternative to TCP stream sockets are UDP datagram sockets. UDP is a thin layer on top of IP, which does not provide data retransmission, data acknowledgment, delivery order, or guaranteed delivery. UDP does handle data integrity by adding a checksum to each packet.

It might seem counterproductive to use TCP for SmartSockets GMD because connections have to send acknowledgment messages in addition to the underlying acknowledgment packets going on with TCP. TCP has been deployed for twenty years, though, and many networking experts have worked very hard to tune TCP for optimal performance. If connections used UDP datagram sockets instead of TCP stream sockets, connections would have to reimplement much of TCP. Using reliable network protocols like TCP simplifies GMD because GMD only has to resend messages when network failures occur. Under normal operation, the simpler GMD connection provides good performance because TCP/IP does the job for which it was designed. When network failures occur, connections can not only detect these failures but also use GMD to totally recover from them.

File-Based and Memory-Based GMD

There are two types of GMD:

- memory-based GMD
- file-based GMD

Memory-based GMD works well for transient network problems, but it does not protect an RTclient from system crashes. Because it stores the messages only in memory, a system crash before the message is delivered can cause the message to be lost.

File-based GMD writes the messages to a file, which can be accessed for re-delivery if there is a system crash. This means file-based GMD is much more reliable, but slower than memory-based GMD. For any software product, performance is slower when data is written to disk frequently and that performance depends on the speed of your local file system. However, writing crucial information, such as a message, to disk is still the best way to ensure system reliability.

When deciding whether to use GMD, you need to decide what is most important for your system, balancing performance and reliability, and determining your tolerance for missed or duplicate messages in the event of network and system crashes.

GMD Composition

Guaranteed message delivery is implemented with several components in messages and connections:

- sequence number message property
- GMD area connection property
- delivery mode message property
- GMD message types
- delivery timeout message property

Most of these components are discussed in detail in Connection Composition on page 71 and Message Composition on page 2. This section gives an overview of how these separate features are integrated in GMD.

Sequence Number

One of the simplest but most important parts of GMD is message sequence numbers. As described in Sequence Number on page 24, the sequence number uniquely identifies the message for GMD so that duplicate messages can be detected by the receiver. Each time a message is sent with GMD, a per-connection outgoing sequence number is incremented, copied to the message sequence number, and saved to the GMD area. Each GMD area also stores the highest sequence number that has been received and acknowledged by this process from each sending process. In the case of a peer-to-peer connection, there is only one sending process.

If recovery is necessary, the sender and receiver can restart exactly where they left off and not use incorrect sequence numbers. Processes performing recovery start with the old sequence numbers to avoid reprocessing messages they have already processed once. This is the main reason that file-based GMD is the recommended type of GMD. Memory-only GMD is useful, though, for small impromptu processes such as prototypes or a debugging session with RTmon.

Note that sequence numbers are not used or needed to detect gaps in streams of messages sent through connections. The underlying reliable network protocols, such as TCP/IP, used by connections already take care of preventing lost data. Connections only need to resend messages for GMD when a network failure occurs.

GMD Area

The GMD area property of a connection holds guaranteed message delivery information for both incoming and outgoing messages. There are two types of GMD:

- file-based GMD

File-based GMD stores the GMD information in files for reliable operation even when network failures occur. Once the data is written to the GMD area files, GMD can recover from many failures to the process, the process's node, or the network (but the files do need to be available for recovery to occur). For example, if a process crashes and is restarted, the restarted process can reopen the file-based GMD area and recover its GMD state, consisting of which messages need to be resent and which messages have already been processed.

- memory-based GMD

Memory-based GMD stores GMD information in a GMD area that is held in memory and is faster than file-based GMD. It protects your messages against network failures and lost connections that do not affect memory. However, if a system failure wipes out memory, such as when a program crashes and restarts, the GMD messages stored in memory in the GMD area are lost.

There is an option, `IPC_GMD_Type`, that sets whether file-based or memory-based GMD is initially attempted.

Sender

When a message is sent with GMD through a connection, the message sequence number is set to an incremented counter, and then a copy of the message is saved in the sender's connection GMD area. The copy is removed when acknowledgment of delivery is received by the sender from the receiving processes.

The sender stores complete messages into the GMD area, which therefore can use large amounts of disk or memory resources if the receiving process falls behind. See [Limiting GMD Resources](#) on page 335 for details on how to constrain GMD resources.

For recovery from network failures, the burden of recovery is on the sender. The sender can reopen the file-based GMD area and simply resend all messages in the GMD area. When messages are resent with GMD, their sequence numbers are not changed. The sender does not have to worry about deciding which message to resend because the receiver discards the duplicate messages that it has already processed.

Receiver

When a GMD message is acknowledged by the receiver, the sequence number of the message is saved in the receiver's connection GMD area as the highest sequence number received. When a resent message is read from a connection, the message sequence number is checked against the highest sequence number in the receiver's connection GMD area. This allows duplicate messages to be detected and discarded.

The receiver only stores highest sequence numbers into the GMD area, which does not usually require much disk or memory resources. RTclient receiver processes store one highest sequence number for each sending RTclient process, however.

Asynchronous Operation For High Performance

Just as operating systems use data buffers and asynchronous techniques to ensure good performance, GMD is generally asynchronous in the sense that processes do not block waiting for GMD operations to complete. Sending processes do not wait for acknowledgment of successful delivery from receiving processes. Most failure notifications (through GMD_FAILURE messages) also occur asynchronously.

Accessing the GMD Area

The GMD area is not directly accessible. The function `TipcConnMsgSend` adds a message to a connection's GMD area. The function `TipcConnRead` removes a message from a connection's GMD area when acknowledgment is received indicating successful delivery. `TipcConnRead` also checks for duplicate messages based on the highest sequence number information stored in the GMD area. The function `TipcConnGmdMsgDelete` removes a message from a connection's GMD area as a result of GMD failure. The function `TipcConnGetGmdNumPending` gets the number of messages within the GMD area. The function `TipcConnGmdResend` reads all messages from the GMD area and resends them. The function `TipcMsgAck` updates the GMD area with highest sequence number information.

Creating the GMD Area

`TipcConnGmdFileCreate` creates the GMD area on disk for file-based GMD. It checks the `IpC_Gmd_Directory` option to determine in what directory to create the GMD area. Each particular GMD area is created once with `TipcConnGmdFileCreate`:

```
if (!TipcConnGmdFileCreate(conn)) {
    /* error */
}
```

Once the GMD area is created, it cannot be changed or destroyed except by destroying the connection. The function `TipcConnSetGmdMaxSize` can be used to set the maximum size (in bytes) of a connection GMD area. See [Limiting GMD Resources](#) on page 335 for more details.

Delivery Mode

As described in [Delivery Mode](#) on page 9, the delivery mode of a message controls what level of guarantee is used when the message is sent through a connection (always with `TipcConnMsgSend`). The available delivery modes are:

T_IPC_DELIVERY_BEST_EFFORT In this mode, no special actions, such as ACKs, are taken to ensure delivery of sent messages. The message is delivered unless network failures or process failures cause the message to be lost. If the message is not delivered, there is no way for the sender to know that delivery failed. When there is a failure, it is possible for some messages to be lost or to be delivered in a different order than they were published.

This is the default mode.

T_IPC_DELIVERY_ORDERED In this mode, no special actions, such as ACKs, are taken to ensure delivery of sent messages. Messages can still be lost in the event of a failure, but this mode ensures that messages are delivered in the order in which they were published. This is useful for applications where order is critical, but the overhead required by GMD results in unacceptable performance degradation.

T_IPC_DELIVERY_SOME

In this mode, the sending process saves a copy of the message in the connection GMD area until the message is successfully delivered, and the sender can also resend the message if necessary. Delivery is considered successful if the sent message is acknowledged by at least one receiving process.

T_IPC_DELIVERY_ALL

In this mode, the sending process saves a copy of the message in the connection GMD area until the message is successfully delivered, and the sender can also resend the message if necessary. Delivery is not considered successful until all receiving processes acknowledge the sent message.

For two processes communicating using a non-RTclient and non-RTserver T_IPC_CONN connection, T_IPC_DELIVERY_SOME and T_IPC_DELIVERY_ALL are identical, because there is only one process receiving the message. For RTclient processes, the two modes do differ if more than one RTclient process is subscribing to the subject in the destination of the message.

GMD Message Types

As described in Acknowledgment of Delivery, GMD needs some form of acknowledgment to know when a message has been successfully delivered. Connections, RTclient, and RTserver use several different message types to implement GMD:

GMD_ACK	Sent by receiver to acknowledge successful GMD.
GMD_FAILURE	Notification of GMD failure (constructed and processed by sender).
GMD_DELETE	Sent by RTclient to notify RTserver to cancel GMD for a message.
GMD_NACK	Sent by RTserver to notify RTclient of certain types of GMD failure.
GMD_STATUS_CALL	Sent by RTclient to query RTserver for GMD status.
GMD_STATUS_RESULT	Sent by RTserver to RTclient with GMD status information.

The message types GMD_DELETE, GMD_NACK, GMD_STATUS_CALL, and GMD_STATUS_RESULT are not used by connection GMD, only by RTclient and RTserver GMD. These message types are discussed in detail in GMD Message Types on page 347.

GMD_ACK

GMD_ACK messages are sent by a receiving process to acknowledge successful delivery of a message with GMD. GMD_ACK messages are sent automatically when a message is destroyed, but can be sent manually instead. GMD_ACK messages are automatically processed by connections so that the SmartSockets programs are not cluttered with having to read and process one GMD_ACK message for each outgoing message sent with GMD.

GMD_FAILURE

GMD handles most network failures, but there are some that GMD cannot overcome on its own, such as a receiving process which goes into an infinite loop. Unlike sockets, which do not provide a way to tell how much data was lost, GMD explicitly notifies a sending process that a GMD failure has taken place. When most types of GMD failure happen, a GMD_FAILURE message is delivered back to the sender process. Each GMD_FAILURE message contains several fields, including the failed message and an error number indicating the type of failure.

For connection GMD, the only GMD_FAILURE error number possible is a delivery timeout, which occurs if a sender does not get acknowledgment of successful delivery within a specified period of time, which is configurable with the message, message type, and connection delivery timeout properties.

A GMD_FAILURE message indicates the message could not be delivered with GMD successfully within the parameters, such as delivery timeout, set by the application. When a GMD failure occurs, it is up to the sender to decide what to do and then take some user-defined action if recovery is feasible. Unfortunately, this level of recovery is very application-specific, and SmartSockets cannot perform it on its own. Recovering from GMD failures is discussed further in *Handling GMD Failures*.

Delivery Timeout

The Delivery Timeout property specifies how long GMD has to deliver a message and it works together with the value set for the Server_Read_Timeout option. The connection delivery timeout property is used as a default for messages with no preset delivery timeout. The delivery timeout is specific to GMD.

Delivery timeouts are checked only when data is received from the RTserver. If no messages are being received, the RTclient uses the value set for Server_Read_Timeout as the interval for sending a keep alive message to the RTserver. When the RTserver replies, then the delivery timeouts are checked. If the delivery timeout is set to a value smaller than the value for Server_Read_Timeout, the actual timeout used is the Server_Read_Timeout because the delivery timeouts are not checked until after the Server_Read_Timeout interval has triggered a keep alive message.

Working With GMD

This section discusses how to configure and use GMD with connections. To learn more about working with the basic features of connections, see [Working With Connections](#) on page 89. The following example programs show the code used to send messages with GMD between two processes through a connection. The programs also show the inner workings of GMD. There are two parts to the example: a server process and a client process.

The source code files for these examples are located in these directories:

UNIX:

```
$RTHOME/examples/smrtsock/manual
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]
```

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

The online source files have additional `#ifdefs` to provide C++ support. These `#ifdefs` are not shown to simplify the example.

Example 30 Common Header Source Code

```
/* connnmd.h -- common header for connections GMD example */

#include <rtworks/ipc.h>

#define GMD_CONN_NAME "tcp:_node:5252"

void create_conn_cb();
```

Example 31 Common Callback Source Code

```

/* conngmd.c -- connections GMD example utilities */

#include "conngmd.h"

/* ===== */
/*..cb_conn_msg -- connection msg callback */
void T_ENTRY cb_conn_msg(
    T_IPC_CONN conn,
    T_IPC_CONN_MSG_CB_DATA data,
    T_CB_ARG arg) /* really (T_STR) */
{
    T_IPC_MT mt;
    T_STR name;
    T_INT4 seq_num;
    T_STR info;

    /* This example callback function is not intended to show any */
    /* useful processing, only how GMD works. */
    TutOut("%s: ", (T_STR)arg);

    /* print out the name of the type of the message */
    if (!TipcMsgGetType(data->msg, &mt)) {
        TutOut("Could not get message type from message: error
<%s>.\n",
            TutErrStrGet());
        return;
    }
    if (!TipcMtGetName(mt, &name)) {
        TutOut("Could not get name from message type: error <%s>.\n",
            TutErrStrGet());
        return;
    }
    TutOut("type %s, ", name);
    /* print out the sequence number of the message to show how inner */
    /* workings of GMD operate */
    if (!TipcMsgGetSeqNum(data->msg, &seq_num)) {
        TutOut("Could not get sequence number from msg: error <%s>.\n",
            TutErrStrGet());
        return;
    }
    TutOut("seq_num %d\n", seq_num);
}

```

```

    /* print the first field of the message (INFO or GMD_ACK) */
    if (mt == TipcMtLookupByNum(T_MT_INFO)) {
        if (!TipcMsgSetCurrent(data->msg, 0)
            || !TipcMsgNextStr(data->msg, &info)) {
            TutOut("Could not get field from INFO message: error
<%s>.\n",
                TutErrStrGet());
            return;
        }
        TutOut("  INFO field: %s\n", info);
    }
    else if (mt == TipcMtLookupByNum(T_MT_GMD_ACK)) {
        if (!TipcMsgSetCurrent(data->msg, 0)
            || !TipcMsgNextInt4(data->msg, &seq_num)) {
            TutOut("Could not get field from GMD_ACK msg: error <%s>.\n",
                TutErrStrGet());
            return;
        }
        TutOut("  GMD_ACK field: %d\n", seq_num);
    }
} /* cb_conn_msg */

/* ===== */
/*..create_conn_cb -- create example callbacks */
void T_ENTRY create_conn_cb(conn)
T_IPC_CONN conn;
{
    /* create callbacks to be executed when certain operations occur */
    TutOut("Create callbacks.\n");

    /* create read callback to show received messages */
    if (TipcConnReadCbCreate(conn, NULL, cb_conn_msg, "read") ==
        NULL) {
        TutOut("Could not create read cb: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    /* create write callback to show sent messages */
    if (TipcConnWriteCbCreate(conn, NULL, cb_conn_msg, "write")
        == NULL) {
        TutOut("Could not create write cb: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    /* create default callback to show processed messages */
    if (TipcConnDefaultCbCreate(conn, cb_conn_msg, "default") ==
        NULL) {
        TutOut("Could not create default cb: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
} /* create_conn_cb */

```

Example 32 Server Source Code

```

/* conngmds.c -- connections GMD example server */
/*
This server process waits for a client to connect to it, creates some callbacks to show how GMD
works, and then loops receiving and processing messages.
*/
#include "conngmd.h"

/* ===== */
/*..accept_client -- accept connection from new client */
T_IPC_CONN accept_client(server_conn)
T_IPC_CONN server_conn;
{
    T_IPC_CONN client_conn; /* connection to client */

    client_conn = TipcConnAccept(server_conn);
    if (client_conn == NULL) {
        TutOut("Could not accept client: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
    create_conn_cb(client_conn); /* to show inner workings of GMD */
    return client_conn;
} /* accept_client */

/* ===== */
/*..main -- main program */
int main()
{
    T_IPC_CONN server_conn; /* used to accept client */
    T_IPC_CONN client_conn; /* connection to client */

    TutOut("Configuring server to use file-based GMD.\n");
    TutCommandParseStr("setopt unique_subject conngmds");

    TutOut("Creating server connection to accept clients on.\n");
    server_conn = TipcConnCreateServer(GMD_CONN_NAME);
    if (server_conn == NULL) {
        TutOut("Could not create server connection: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
    /* accept one client */
    TutOut("Waiting for client to connect.\n");
    client_conn = accept_client(server_conn);

    /* destroy server conn: it's not needed anymore */
    TutOut("Destroying server connection.\n");
    if (!TipcConnDestroy(server_conn)) {
        TutOut("Could not destroy server connection: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
}

```

```

    TutOut("Delete old GMD files.\n");
    if (!TipcConnGmdFileDelete(client_conn)) {
        TutOut("Could not delete old GMD files: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
    TutOut("Read and process messages.\n");
    if (!TipcConnMainLoop(client_conn, T_TIMEOUT_FOREVER)
        && TutErrNumGet() != T_ERR_EOF) {
        TutOut("Could not read and process messages: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
    if (!TipcConnDestroy(client_conn)) {
        TutOut("Could not destroy client connection: error <%s>.\n",
            TutErrStrGet());
    }
    TutOut("Server process exiting successfully.\n");
    return T_EXIT_SUCCESS; /* all done */
} /* main */

```

Example 33 Client Source Code

```

/* conngmdc.c -- connections GMD example client */

/*
The client process connects to the server process and sends two
messages with GMD to the server.
*/

#include "conngmd.h"

/* ===== */
/*..connect_to_server -- create connection to server process */
T_IPC_CONN connect_to_server()
{
    T_IPC_CONN conn; /* connection to server */

    conn = TipcConnCreateClient(GMD_CONN_NAME);
    if (conn == NULL) {
        TutOut("Could not create connection to server: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    create_conn_cb(conn); /* to show inner workings of GMD */

    return conn;
} /* connect_to_server */

```



```

/* ===== */
/*..main -- main program */
int main()
{
    T_IPC_CONN conn; /* connection to server */
    T_IPC_MT mt; /* message type for messages */
    T_IPC_MSG msg; /* message to send */
    T_INT4 num_pending; /* number of messages still pending */

    TutOut("Configuring client to use file-based GMD.\n");
    TutCommandParseStr("setopt unique_subject conngmdc");

    TutOut("Creating connection to server process.\n");
    conn = connect_to_server();

    if (!TipcConnSetTimeout(conn, T_IPC_TIMEOUT_DELIVERY, 1.5)) {
        TutOut("Could not set conn delivery timeout: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    TutOut("Delete old GMD files.\n");
    if (!TipcConnGmdFileDelete(conn)) {
        TutOut("Could not delete old GMD files: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    TutOut("Create GMD area.\n");
    if (!TipcConnGmdFileCreate(conn)) {
        TutOut("Could not create GMD area: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    TutOut("Constructing and sending an INFO message.\n");
    mt = TipcMtLookupByNum(T_MT_INFO);
    if (mt == NULL) {
        TutOut("Could not look up INFO message type: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    msg = TipcMsgCreate(mt);
    if (msg == NULL) {
        TutOut("Could not create INFO message: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    if (!TipcMsgSetDeliveryMode(msg, T_IPC_DELIVERY_ALL)) {
        TutOut("Could not set message delivery mode: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
}

```

```

if (!TipcMsgAppendStr(msg, "GMD test #1")) {
    TutOut("Could not append field to INFO message: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

if (!TipcConnMsgSend(conn, msg)) {
    TutOut("Could not send INFO message: error <%s>.\n",
           TutErrStrGet());
}

if (!TipcMsgDestroy(msg)) {
    TutOut("Could not destroy message: error <%s>.\n",
           TutErrStrGet());
}

/* use convenience functions this time */
TutOut("Send another INFO message with GMD.\n");
if (!TipcConnMsgWrite(conn, mt,
                      T_IPC_PROP_DELIVERY_MODE,
                      T_IPC_DELIVERY_ALL,
                      T_IPC_FT_STR, "GMD test #2",
                      NULL)) {
    TutOut("Could not send 2nd INFO message: error <%s>.\n",
           TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
TutOut("Read data until all acknowledgments come in.\n");
do {
    if (!TipcConnMainLoop(conn, 1.0)) {
        TutOut("Could not read from conn: error <%s>.\n",
               TutErrStrGet());
        break;
    }
    if (!TipcConnGetGmdNumPending(conn, &num_pending)) {
        TutOut("Could not get pending message count: error <%s>.\n",
               TutErrStrGet());
        break;
    }
} while (num_pending > 0);

if (!TipcConnDestroy(conn)) {
    TutOut("Could not destroy connection: error <%s>.\n",
           TutErrStrGet());
}
TutOut("Client process exiting successfully.\n");
return T_EXIT_SUCCESS; /* all done */
} /* main */

```

Compiling, Linking, and Running

To compile, link, and run the example programs, first you must either copy the programs to your own directory or have write permission in these directories:

UNIX:

```
$RTHOME/examples/smrtsock/manual
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]
```

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

Step 1 Compile and link the programs

UNIX:

```
$ rtlink -o conngmds.x conngmds.c conngmdu.c
$ rtlink -o conngmdc.x conngmdc.c conngmdu.c
```

OpenVMS:

```
$ cc conngmds.c, conngmdc.c, conngmdu.c
$ rtlink /exec=conngmds.exe conngmds.obj, conngmdu.obj
$ rtlink /exec=conngmdc.exe conngmdc.obj, conngmdu.obj
```

Windows:

```
$ nmake /f cngsw32m.mak
$ nmake /f cngcw32m.mak
```

To run the programs, start the server process first in one terminal emulator window and then the client process in another terminal emulator window or in the background (batch).

Step 2 Start the server program in the first window

UNIX:

```
$ conngmds.x
```

OpenVMS:

```
$ run conngmds.exe
```

Windows:

```
$ conngmds.exe
```

Step 3 Start the client program in the second window**UNIX:**

```
$ conngmdc.x
```

OpenVMS:

```
$ run conngmdc.exe
```

Windows:

```
$ conngmdc.exe
```

This is a sample of the output from the server program:

```
Configuring server to use file-based GMD.
Creating server connection to accept clients on.
Waiting for client to connect.
Create callbacks.
Destroying server connection.
Delete old GMD files.
Read and process messages.
read: type info, seq_num 1
  INFO field: GMD test #1
read: type info, seq_num 2
  INFO field: GMD test #2
default: type info, seq_num 1
  INFO field: GMD test #1
write: type gmd_ack, seq_num 0
  GMD_ACK field: 1
default: type info, seq_num 2
  INFO field: GMD test #2
write: type gmd_ack, seq_num 0
  GMD_ACK field: 2
Server process exiting successfully.
```

This a sample of the output from the client program:

```
Configuring client to use file-based GMD.
Creating connection to server process.
Create callbacks.
Delete old GMD files.
Create GMD area.
Constructing and sending an INFO message.
write: type info, seq_num 0
  INFO field: GMD test #1
Send another INFO message with GMD.
write: type info, seq_num 0
  INFO field: GMD test #2
Read data until all acknowledgments come in.
read: type gmd_ack, seq_num 0
  GMD_ACK field: 1
read: type gmd_ack, seq_num 0
  GMD_ACK field: 2
Client process exiting successfully.
```

Configuring GMD

If you want to use the default configuration for GMD, there is little configuration required. File-based GMD, rather than memory-only GMD, is the recommended way to use GMD because of its ability to recover from process failures. The main configuration required for file-based GMD is to set the option `Unique_Subject` to a value other than the default, to a unique and consistent name that no other process on the node is using.

To configure GMD, you set the GMD-related options using the `setopt` command. Many of these options are not required unless you want to use a value other than the default setting. The GMD options for peer-to-peer connections are the same as the GMD options for RTclients. For more detailed information about the options, see Chapter 8, Options Reference.

Here are the options you can set to change your GMD configuration:

<code>Ipc_Gmd_Auto_Ack</code>	specifies whether automatic acknowledgment of received GMD messages is enabled or disabled.
<code>Ipc_Gmd_Auto_Ack_Policy</code>	specifies when a received GMD message is automatically acknowledged. Automatic acknowledgement must be enabled.
<code>Ipc_Gmd_Directory</code>	specifies the location of the GMD area on disk for file-based GMD. This option does not apply to memory-based GMD. The default values are: <ul style="list-style-type: none"> • UNIX: <code>/tmp/rtworks</code> • Windows: <code>%TEMP%\rtworks</code> • OpenVMS: <code>sys\$scratch</code>
<code>Ipc_Gmd_Type</code>	specifies whether file-based or memory-based GMD is first attempted. The default value is <code>default</code> , which means file-based GMD is attempted. Or you can specify <code>memory</code> , which means memory-based GMD is used regardless of the value set for <code>Unique_Subject</code> .
<code>Server_Delivery_Timeout</code>	specifies the time in seconds that the sending process waits for all receiving processes to acknowledge delivery of a guaranteed message. If you set the value to <code>0.0</code> , this option is disabled, and the sending process never times out. The default is 30 seconds.
<code>Unique_Subject</code>	specifies the unique subject to use. For file-based GMD, you must set this to a value other than the default of <code>_Node_Pid</code> .

Notes on File-Based GMD

- File-based GMD is only attempted if both are true:
 - `Unique_Subject` is set to a value other than its default, and
 - `IPC_GMD_Type` is set to `default`

If `IPC_GMD_Type` is set to `memory`, memory-based GMD occurs even if you set a value for `Unique_Subject`. However, if you did not set a value for `Unique_Subject`, memory-based GMD is used even if `IPC_GMD_Type` is set to `default`.

- When `Unique_Subject` is set, and `IPC_GMD_Type` is `default`, SmartSockets attempts file-based GMD, but sometimes must revert to memory-based GMD. See Reverting to Memory-Based GMD on page 333.
- Although you specify file-based GMD, memory-based GMD is used whenever file-based GMD is attempted unsuccessfully. This provides a measure of safety, because even though file-based GMD might fail, your messages are still protected under GMD.
- The default value for `Unique_Subject` is `_Node_Pid`, where *Node* is the network node name of the computer on which the process is running, and *Pid* is the operating system process identifier of the process. The default `_Node_Pid` usually changes if the program is restarted, because of the new value for *Pid*. This prevents the restarted program from finding the old GMD files, and is why `Unique_Subject` must be explicitly set to use file-based GMD.

For RTclient and RTserver processes, the option `Unique_Subject` must be set to an RTserver group-wide unique name. Keep in mind that for connection programs, `Unique_Subject` does not really specify a subject, only a unique name for GMD.

For example:

```
TutCommandParseStr("setopt unique_subject conngmdc");
```

Reverting to Memory-Based GMD

When `Unique_Subject` is set, and `Ipc_Gmd_Type` is `default`, `SmartSockets` attempts file-based GMD. If file-based GMD cannot be carried out, `SmartSockets` reverts to memory-based GMD for the message. Generally, when `SmartSockets` reverts from file-based to memory-based GMD, it is because the file-based GMD area could not be written to, for example, due to a permissions problem.

If file-based GMD cannot be used, a warning is issued and the process switches to memory-only GMD. For example:

```
WARNING: Could not create GMD file(s) with base name
<tcp__node_5252_conngmdc>
for connection <server:tcp:_node:5252:1>.
The option Unique_Subject must be set to a value other than the
default (_himalia.talarian.com_10783).
WARNING: Switching to memory-only GMD.
```

To check if a process is properly configured for file-based GMD, use the function `TipcGetGmdDir`. For example:

```
if (TipcGetGmdDir() == NULL) {
    /* process is not configured properly */
}
```

`TipcGetGmdDir` attempts to create the needed directories if they do not exist.

Deleting Files From an Old GMD Area

When a process creates a connection, the process may not want to use the file-based GMD information from a previous connection with the same GMD configuration. Some applications, for example, may want to have the concept of a complete restart where all old GMD information is purged. The function `TipcConnGmdFileDelete` can be used to delete the old GMD files. For example:

```
if (!TipcConnGmdFileDelete(conn)) {
    /* error */
}
```

When a process using GMD creates a client connection with `TipcConnCreateClient` or accepts a connection from a client with `TipcConnAccept`, it typically does one of two things:

- discards old GMD information by calling `TipcConnGmdFileDelete`
- resends old GMD information by calling `TipcConnGmdResend` (see [Resending Messages](#) on page 339)

Creating a GMD Area

Before any messages can be sent or received with GMD, a GMD area must be created with the function `TipcConnGmdFileCreate`. For example:

```
if (!TipcConnGmdFileCreate(conn)) {
    /* error */
}
```

If the GMD area files already exist on disk, `TipcConnGmdFileCreate` reads in the contents of the old GMD area. The functions `TipcConnRead` and `TipcConnMsgSend` automatically call `TipcConnGmdFileCreate` to create a GMD area for a connection that needs one. For example, the server example program `conngmds.c` does not explicitly call `TipcConnGmdFileCreate`, but instead lets `TipcConnMainLoop`, which eventually calls `TipcConnRead`, take care of it. For most applications, both ways are equivalent.

Specifying Ipc_Gmd_Directory

File-based GMD places all files it creates under the directory specified in the option `Ipc_Gmd_Directory`, which defaults to these locations:

UNIX:

```
/tmp/rtworks
```

OpenVMS:

```
sys$scratch
```

Windows:

```
%TEMP%\rtworks
```

The directory specified in `Ipc_Gmd_Directory` should be on a local file system for best performance. File-based GMD also uses `Unique_Subject` to generate pathnames for the GMD area. See the reference page for `TipcConnGmdFileCreate` in the *TIBCO SmartSockets Application Programming Interface* reference for full details on GMD area pathnames.

Limiting GMD Resources

The resources used by GMD can be constrained by storage used, by elapsed time, or by both.

GMD Area Maximum Size

A connection GMD area uses disk files or memory to store the information needed for GMD. A GMD area has a maximum size (in bytes) that can be used to limit the amount of disk space or memory that a GMD area can use. The default connection GMD area maximum size is 0, which means that no maximum size limit checking is performed. Use the `TipcConnGetGmdMaxSize` function to get the GMD area maximum size. For example:

```
if (!TipcConnGetGmdMaxSize(conn, &gmd_max_size)) {
    /* error */
}
```

The function `TipcConnSetGmdMaxSize` is used to set the GMD area maximum size. For example:

```
/* limit GMD area to one megabyte */
if (!TipcConnSetGmdMaxSize(conn, 1000000)) {
    /* error */
}
```

If the connection GMD area maximum size is exceeded, then no further messages can be sent with GMD until some unacknowledged, previously-sent messages are acknowledged. This is one of the few synchronous notifications of a GMD-related failure, compared to the asynchronous `GMD_FAILURE` messages.

Delivery Timeout

The connection delivery timeout property can be used to limit the amount of time that GMD has to deliver each message. This can be useful for real-time applications that need to be notified if a certain action does not take place within a certain period of time. If a different timeout is required for an individual message, the message's delivery timeout property can instead be set directly.

Sending Messages

Sending messages with GMD is very easy. The message delivery mode must first be set to `T_IPC_DELIVERY_ALL` or `T_IPC_DELIVERY_SOME`, and then the message can be sent as usual with `TipcConnMsgSend`. For example:

```
if (!TipcMsgSetDeliveryMode(msg, T_IPC_DELIVERY_ALL)) {
    /* error */
}
if (!TipcConnMsgSend(conn, msg)) {
    /* error */
}
```

`TipcConnMsgSend` automatically calls `TipcConnGmdFileCreate` if necessary. For GMD, `TipcConnMsgSend` increments an internal per-connection outgoing sequence number, sets the message sequence number to the incremented value, saves a copy of the message in the connection GMD area, and saves the current wall clock time in the GMD area (for detecting a delivery timeout). Note that the message sequence number is set after the connection write callbacks are called, as illustrated by this output from the example client program (the sequence number is zero in the write callback):

```
Constructing and sending an INFO message.
write: type info, seq_num 0
      INFO field: GMD test #1
Send another INFO message with GMD.
write: type info, seq_num 0
      INFO field: GMD test #2
```

If all outgoing messages of a certain type need to be sent with GMD, the function `TipcMtSetDeliveryMode` can be used once instead of calling `TipcMsgSetDeliveryMode` for each message of that type.

Receiving Messages

Receiving messages with GMD is even easier than sending messages with GMD; no code changes are needed. `TipcConnRead` automatically calls `TipcConnGmdFileCreate` if necessary. For GMD, `TipcConnRead` recognizes a message resent with GMD (through an internal message resend property) and checks if the resent message has a sequence number lower than the highest sequence number already acknowledged from the sending process. The check also handles long-running processes that may overflow and wrap around the four-byte sequence number. If the resent message has already been acknowledged, then `TipcConnRead` immediately destroys the message, which acknowledges it again so that the sender is notified this time of successful delivery.

`TipcConnRead` ignores any duplicate (resent) messages that are sent to it while the application is processing the message and has not yet sent an acknowledgement. This means that `TipcConnRead` keeps all received but unacknowledged GMD messages, and discards any subsequent GMD messages with the same sequence number and from the same publisher as long as the original GMD message remains unacknowledged. All other non-resent GMD messages are allowed to pass through regardless of their sequence number. This allows flexibility and correct behavior when some processes use `TipcConnGmdFileDelete` and others do not (thus continuing to use old sequence numbers).

`TipcConnRead` also handles `GMD_ACK` messages directly so that the application code never has to worry about taking care to read and process one `GMD_ACK` message for each outgoing message sent with GMD. When a `GMD_ACK` message is received, the corresponding message is removed from the connection GMD area. The following output from the example client program shows how `GMD_ACK` messages are visible to the connection read callbacks but not to the connection process or default callbacks:

```
Read data to allow acknowledgments time to come in.
read: type gmd_ack, seq_num 0
      GMD_ACK field: 1
read: type gmd_ack, seq_num 0
      GMD_ACK field: 2
```

Acknowledging Messages

Acknowledging messages with GMD is also very easy. `TipcMsgDestroy` automatically calls `TipcMsgAck` to acknowledge the message for GMD. `TipcMsgAck` can also be called manually to acknowledge a message. For example:

```
if (!TipcMsgAck(msg)) {
    /* error */
}
```

`TipcMsgAck` constructs a `GMD_ACK` message containing the sequence number of the message to be acknowledged and sends the `GMD_ACK` message through the connection that the message to be acknowledged was received on.

`TipcMsgAck` knows which connection to use through an internal message property containing the connection the message was received on.

You can disable the automatic acknowledgement of GMD messages with the `Ipc_Gmd_Auto_Ack` option. When automatic acknowledgment is disabled, it becomes the user's responsibility to acknowledge receipt of the GMD message by calling `TipcMsgAck`.

Waiting for Completion of GMD

GMD senders must read messages occasionally to receive the acknowledgments. If a connection process both sends and receives messages at regular intervals, no extra actions are needed because the acknowledgments travel with the normal flow of messages. A short-running or sending-only process can accomplish this by calling `TipcConnMainLoop` or `TipcConnRead` before the program exits. A sending process can also check how many outgoing GMD messages are still pending with `TipcConnGetGmdNumPending`. This is useful for waiting until all acknowledgments arrive. For example:

```
TutOut("Read data until all acknowledgments come in.\n");
do {
    if (!TipcConnMainLoop(conn, 1.0)) {
        /* error */
        break;
    }
    if (!TipcConnGetGmdNumPending(conn, &num_pending)) {
        /* error */
        break;
    }
} while (num_pending > 0);
```

Resending Messages

If a program detects an error, such as `TipcConnMainLoop` or `TipcConnFlush` fails, recovery is typically initiated by creating a new connection and then calling `TipcConnGmdResend` to resend the old GMD messages. For example:

```
if (!TipcConnFlush(conn)) {
    TutOut("Lost connection to server process.\n");
    /* destroy old connection and create new one */
    if (!TipcConnDestroy(conn)) {
        /* error */
    }
    conn = connect_to_server();
    /* resend messages from GMD area (N/A for memory-only GMD) */
    if (!TipcConnGmdResend(conn)) {
        /* error */
    }
}
```

`TipcConnGmdResend` automatically calls `TipcConnGmdFileCreate` if necessary. `TipcConnGmdResend` reads all messages from the connection GMD area, preserves their sequence numbers, marks them as being resent (through an internal message resend property), and sends them with `TipcConnMsgSend`. Resending GMD messages does not change their timestamps used to detect delivery timeouts; this prevents repetitive resends from hiding delivery timeout detection. Note that with memory-only GMD there can be no resending, as the GMD area is lost when the old connection is destroyed.

When a process using GMD creates a client connection with `TipcConnCreateClient` or accepts a connection from a client with `TipcConnAccept`, it typically does one of two things:

- discards old GMD information by calling `TipcConnGmdFileDelete` (see [Deleting Files From an Old GMD Area on page 333](#)), or
- resends old GMD information by calling `TipcConnGmdResend`

Both the publishing client and the RTserver have a GMD area and GMD messages are stored in both. If the RTserver goes down, the publishing client has all the GMD messages in its own GMD area and can resend those messages when it connects back into the RTserver cloud.

Receiving Duplicate Messages

As described in [Receiving Messages on page 337](#) and [Resending Messages on page 339](#), when duplicate messages are received, they are always messages that have been resent. Resent messages always have their sequence numbers checked against the highest sequence number stored in the connection GMD area.

In the previous example, if the server program was sent the two INFO messages twice, the second set of messages produces this output:

```
read: type info, seq_num 1
      INFO field: GMD test #1
write: type gmd_ack, seq_num 0
      GMD_ACK field: 1
read: type info, seq_num 2
      INFO field: GMD test #2
write: type gmd_ack, seq_num 0
      GMD_ACK field: 2
```

Note how acknowledgments are sent immediately to notify the sender that the receiver has already processed these messages.

Handling GMD Failures

With connections there are two types of GMD-related failures possible:

- A synchronous error such as `TipcConnFlush` returning `FALSE` and setting the global `SmartSockets` error number to `T_ERR_EOF` or `T_ERR_FAILURE_DETECTED`. These errors are easy to detect and handle, such as call `TipcConnDestroy` and then `TipcConnCreateClient`.
- An asynchronous `GMD_FAILURE` error such as a delivery timeout. The causes of these errors are very application-specific.

As discussed in `GMD_FAILURE` on page 321, recovery from `GMD_FAILURE` messages is very application-specific, and `SmartSockets` cannot perform it on its own. The `GMD_FAILURE` message notifies the process that there is a problem, and the process can take whatever user-defined action is necessary. `SmartSockets` by default outputs a warning, terminates GMD for the failed message, and continues.

GMD_FAILURE Messages

When GMD fails asynchronously, a `GMD_FAILURE` message is created internally by `SmartSockets` and `TipcConnMsgProcess` is called to process the message and thus notify the sender that there has been a GMD failure. GMD programs can create connection process callbacks for the `GMD_FAILURE` message type to execute their own recovery procedures. The failed message is left in the connection GMD area, and it is up to the `GMD_FAILURE` process callbacks to delete the message (and thus terminate GMD for that message) or resend the message.

Each `GMD_FAILURE` message contains four fields:

- a `MSG` message field containing the message sent by this process where GMD failed
- a `STR` string field containing the name of the receiving process where GMD failed (actually the value of the receiving process's `Unique_Subject` option)
- an `INT4` integer field containing a `SmartSockets` error number describing how GMD failed
- a `REAL8` numeric field containing the wall clock time the failed message was originally sent

Delivery Timeout Failures

As described in GMD_FAILURE on page 321, the only type of GMD_FAILURE message produced for non-RTclient/non-RTserver GMD is a delivery timeout failure: the third field of a GMD_FAILURE message is T_ERR_GMD_SENDER_TIMEOUT. Connections automatically check for delivery timeouts whenever data is read from the connection with TipcConnRead, or the connection is checked to see if data can be read with TipcConnCheck. Thus it is important to use TipcConnRead and TipcConnCheck frequently enough and with *timeout* parameters that are not larger than the connection delivery timeout property or the delivery timeout of an individual message.

Default Processing of GMD_FAILURE Messages

All connections by default have a process callback for GMD_FAILURE messages that use TipcCbConnProcessGmdFailure as the callback function. TipcCbConnProcessGmdFailure outputs a warning and deletes the message from the connection GMD area.

TipcCbConnProcessGmdFailure is intended as a sample callback that is designed only to warn the user that guaranteed delivery of a message has failed. More sophisticated applications should destroy the callback that uses TipcCbConnProcessGmdFailure and create their own process callbacks for GMD_FAILURE messages to perform actions such as user-defined recovery procedures. See the reference page for TipcCbConnProcessGmdFailure in the *TIBCO SmartSockets Application Programming Interface* reference for full details on this callback.

When a GMD_FAILURE message is processed by a sender, one of two actions is typically performed:

- the sender process does not want to resend the message, and thus uses TipcConnGmdMsgDelete to terminate GMD for the message
- the sender process does want to resend the message, and thus takes some kind of user-defined action to correct the problem and then uses TipcConnGmdMsgResend to resend the message

Resending a Message

A single message can be resent with TipcConnGmdMsgResend. For example:

```
if (!TipcConnGmdMsgResend(conn, msg)) {
    /* error */
}
```

TipcConnGmdMsgResend usually is only used from a GMD_FAILURE connection process callback.

Deleting a Message

A single message can be deleted from the connection GMD area with `TipcConnGmdMsgDelete`. For example:

```
if (!TipcConnGmdMsgDelete(conn, msg)) {
    /* error */
}
```

`TipcConnGmdMsgDelete` usually is only used from a `GMD_FAILURE` connection process callback.

Limitations of GMD

GMD can recover from most network failures, but there are a few scenarios to be aware of when using GMD:

- If a sending process crashes before an outgoing message can be completely saved to the GMD area, the message cannot be recovered.
- If a receiving process crashes after processing a message but before the highest sequence number can be updated in the GMD area, the message can potentially be processed twice.
- Mixing GMD and message priorities can cause the highest sequence number in the GMD area to be updated in non-sequential order. If the receiving process crashes while processing messages in non-increasing sequence number order, the resent messages can potentially be skipped.



It is highly recommended that all GMD messages sent from the same publisher have the same priority setting, or they can be read out of order. Messages read out of order can result in message loss.

These conditions are very unlikely, but can happen if a node or disk crashes at exactly the wrong time.

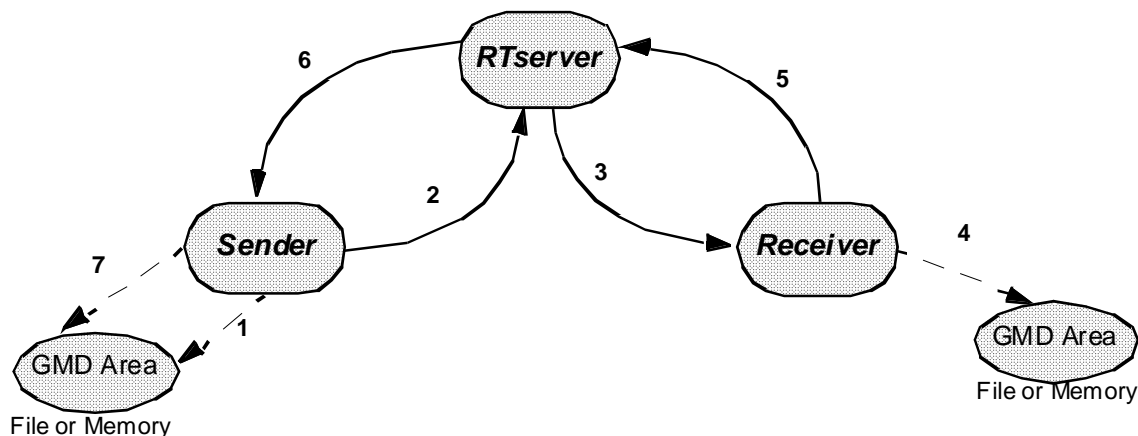
Publish-Subscribe and GMD

RTserver and RTclient can take full advantage of the GMD of connections. In addition to the connection GMD capabilities, both RTclient and RTserver have several additional features related to GMD.

- Delivery can be guaranteed to all subscribing RTclients or to at least one subscribing RTclient.
- RTserver manages the gathering of acknowledgments from the necessary subscribers and sends a single acknowledgment back to the publisher.
- RTclient automatically resends unacknowledged GMD messages when it connects or reconnects to RTserver.
- RTserver can maintain warm RTclient information to prevent GMD messages from being lost if an RTclient crashes and restarts.
- RTserver keeps GMD messages in memory so it can easily resend them without having to notify or wait for the original sending RTclient.
- RTclient can query RTserver for the delivery status of a message sent with GMD.

Figure 34 illustrates the successful processing of a guaranteed message.

Figure 34 Steps Involved in GMD Successful Delivery



1. Message is saved to GMD area.
2. Message is sent to RTserver.
3. Message is sent to Receiver.
4. After processing message, highest sequence number is updated in the receiver's GMD area.
5. Acknowledgment message is sent to RTserver.
6. Acknowledgment message is sent to Sender.
7. Message is deleted from GMD area.

Working with GMD in RTserver and RTclient is very similar to working with GMD in peer-to-peer connections, and thus no complete source code example is shown. The source code files `rtclgmds.c` and `rtclgmldr.c` illustrate the most probable GMD failures and are located in these directories:

UNIX:

`$RTHOME/examples/smrtsock/manual`

OpenVMS:

`RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]`

Windows:

`%RTHOME%\examples\smrtsock\manual`

Delivery Mode in Publish-Subscribe Model

As discussed in Delivery Mode on page 318, in peer-to-peer connections the GMD delivery modes `T_IPC_DELIVERY_SOME` and `T_IPC_DELIVERY_ALL` are the same because there is only one receiving process. With the RTserver and RTclient GMD, `T_IPC_DELIVERY_SOME` is a subset of `T_IPC_DELIVERY_ALL`. With `T_IPC_DELIVERY_ALL`, delivery is not considered successful until all subscribing RTclients acknowledge delivery. With `T_IPC_DELIVERY_SOME`, delivery is considered successful as soon as at least one subscriber acknowledges delivery. This is useful for programs where it is acceptable to continue after at least one subscriber, such as an operator console, receives notification of an event.

Warm RTclient in RTserver

A warm RTclient in RTserver is somewhat analogous to a warm connection to RTserver in RTclient, although there are many differences. As described in Warm Connection to RTserver on page 235, a warm connection to RTserver is a subset of a full connection that allows messages, including messages sent without GMD, to be buffered for later sending.

A warm RTclient in RTserver, however, is used only for GMD. With a warm RTclient, the only thing RTserver remembers is the name of the RTclient and the GMD-using subjects to which the RTclient was subscribing when that RTclient disconnected. RTserver then tracks the GMD messages that this warm RTclient should receive and acknowledge (see How GMD Works in RTserver for a full discussion of how RTserver operates for GMD). RTserver does not buffer any non-GMD messages for the RTclient (those with a delivery mode of `T_IPC_DELIVERY_BEST_EFFORT` or `T_IPC_DELIVERY_ORDERED`).

RTclient informs RTserver to keep warm RTclient information for itself by setting the `Server_Disconnect_Mode` option to `warm` before calling either `TipcSrvCreate` or `TipcSrvDestroy` (both of these functions send the value of `Server_Disconnect_Mode` to RTserver). In this `warm` mode, if an RTclient disconnects for any reason, such as crashes or simply calls `TipcSrvDestroy`, all necessary RTservers (those with direct GMD publishing RTclients) keep warm RTclient information.

The warm RTclient is not considered to be associated with any RTserver, and it can later reconnect to any RTserver in the same multiple RTserver group. Until the warm RTclient reconnects or the timeout specified in the RTserver option `Client_Reconnect_Timeout` is reached, each RTserver continues to buffer GMD messages sent by its own direct RTclients that have a destination subject being subscribed to by the warm RTclient. If the warm RTclient reconnects in time, then all RTservers resend the proper GMD messages to the reconnected RTclient in the proper order. RTclient can even switch from one RTserver to another, and the RTserver takes care of all the necessary rerouting for GMD.

Note that no warm RTserver-to-RTserver information is needed for GMD. An RTserver does route the messages for GMD, but it is the subjects to which the RTclient is subscribing that are really important. Therefore, warm RTclient information is needed in RTserver for GMD, but not warm RTserver information.

GMD Message Types

There are six message types used to implement GMD in connections, RTservers, and RTclients. GMD_ACK and GMD_FAILURE are used in all GMD, while the other four are used only by an RTserver and RTclient. This section describes how these message types are used in RTclient/RTserver GMD.

GMD_ACK

GMD_ACK messages are used by RTclient and RTserver much as they are by peer-to-peer connections. When an RTclient publishes a message with GMD, the subscribing RTclients all send a GMD_ACK message to acknowledge delivery. The original publishing RTclient does not get these acknowledgments, however. RTserver manages the gathering of acknowledgments from the necessary subscribers and sends a single acknowledgment back to the original publisher. This greatly simplifies GMD for the publishing RTclient.

GMD_DELETE

The function TipcSrvGmdMsgServerDelete sends a GMD_DELETE message to RTserver to inform RTserver to terminate GMD for a specific message, which allows RTserver to reclaim the memory for the message. TipcSrvGmdMsgServerDelete is usually only used from a connection process callback for a GMD_FAILURE message, but it can be called at any time by advanced programs that wish to poll for GMD status.

GMD_FAILURE

GMD_FAILURE messages are used to notify RTclient of asynchronous GMD failures much as they are for peer-to-peer connections. In addition to the T_ERR_GMD_SENDER_TIMEOUT error code described in Delivery Timeout Failures on page 342, RTclients should be prepared to handle these additional error codes:

Table 14 GMD Failure Error Numbers

Error Number	Description
T_ERR_GMD_RECEIVER_TIMEOUT	A receiving RTclient (whose Server_Disconnect_Mode option is set to warm) disconnects before acknowledging the message and does not reconnect to RTserver in time.

Table 14 GMD Failure Error Numbers

Error Number	Description
T_ERR_GMD_RECEIVER_EXIT	A receiving RTclient (whose Server_Disconnect_Mode option is set to gmd_failure) disconnects before acknowledging the message.
T_ERR_GMD_NO_RECEIVERS	No RTclients are subscribing to the destination subject of the message, and the option Zero_Recv_Gmd_Failure in RTserver is set to TRUE.

GMD_NACK

GMD_NACK messages are sent from RTserver to RTclient, indicating that a GMD failure occurred. RTclient automatically builds a GMD_FAILURE message from the information in the GMD_NACK message and processes the GMD_FAILURE message. RTclients should not try to process GMD_NACK messages and should only manipulate callbacks for GMD_FAILURE messages. The GMD_NACK message type is documented for completeness and to illustrate how GMD operates.

GMD_STATUS_CALL

The function TipcSrvGmdMsgStatus sends a GMD_STATUS_CALL message to RTserver to poll RTserver for the current GMD status of a specific message. If RTserver is aware of the message, it sends back a GMD_STATUS_RESULT message with the requested information. TipcSrvGmdMsgStatus is intended to be used from a connection process callback for a GMD_FAILURE message.

GMD_STATUS_RESULT

GMD_STATUS_RESULT messages are sent from RTserver to RTclient as the result of a query with TipcSrvGmdMsgStatus. Each GMD_STATUS_RESULT message contains these four fields:

- an INT4 integer field containing the sequence number property of the message
- a STR_ARRAY string array field containing the names of the RTclients that have acknowledged successful delivery of the message
- a STR_ARRAY string array field containing the names of the RTclients where GMD failed for the message
- a STR_ARRAY string array field containing the names of the RTclients where delivery is still pending for the message

The polling RTclient acquires the GMD_STATUS_RESULT message after calling TipcSrvGmdMsgStatus, using functions like TipcSrvMsgSearchType.

RTclient GMD Considerations

Configuring RTclient for GMD

The configuration of GMD for the RTclient is similar to the configuration for peer-to-peer connections. The same options apply:

- Ipc_Gmd_Auto_Ack
- Ipc_Gmd_Auto_Ack_Policy
- Ipc_Gmd_Directory
- Ipc_Gmd_Type
- Server_Delivery_Timeout
- Unique_Subject

The settings for these options have the same effect for GMD with RTclients as for GMD with peer-to-peer connections. For an RTclient, you can specify the values for the options in the RTclient’s startup command file or by calling the API function TutCommandParseStr, or by calling one of the TutOptionSetType API functions.

For more information, see Configuring GMD on page 331.

In addition to these options, you can also specify the option Server_Disconnect_Mode for the RTclient. The Server_Disconnect_Mode option specifies the action RTserver should take when RTclient disconnects from RTserver. The possible values are:

warm	RTserver saves subject information about RTclient for guaranteed message delivery so that no messages are lost.
gmd_failure	RTserver destroys all information about RTclient and causes pending guaranteed message delivery to fail.
gmd_success	RTserver destroys all information about RTclient and causes pending guaranteed message delivery to succeed.

Setting Server_Disconnect_Mode to warm is useful when RTclient must run continuously and not lose any messages even if it crashes or accidentally terminates. In this mode, RTserver remembers the subjects being subscribed to by the disconnecting RTclient and buffers guaranteed messages. When an RTclient with the same value for the option Unique_Subject reconnects to RTserver, RTserver resends the guaranteed messages to RTclient. The option Client_Reconnect_Timeout in RTserver controls the maximum amount of time (in

seconds) RTclient has to reconnect. If RTclient does not reconnect to RTserver within `Client_Reconnect_Timeout` seconds, then RTserver clears the guaranteed messages that have not been acknowledged by this RTclient and sends a `GMD_NACK` message back to the sender of these messages.

Setting `Server_Disconnect_Mode` to `gmd_failure` is useful for short-lived operation. In this mode, RTserver clears the guaranteed messages that have not been acknowledged by this RTclient and sends a `GMD_NACK` message back to the sender of these messages.

Setting `Server_Disconnect_Mode` to `gmd_success` is useful for short-lived operation or when RTclient wants to exit cleanly without causing GMD failure in the publishing process. In this mode, RTserver clears the guaranteed messages that have not been acknowledged by this RTclient and sends a `GMD_ACK` message back to the sender of these messages.

DISCONNECT Message Type

When RTclient creates a full connection to RTserver by calling `TipcSrvCreate(T_IPC_SRV_CONN_FULL)`, the value of the RTclient option `Server_Disconnect_Mode` is sent to RTserver in a `CONNECT_CALL` message. `Server_Disconnect_Mode` specifies the action RTserver should take when RTclient disconnects from RTserver. To allow RTclient to change `Server_Disconnect_Mode` before disconnecting, the function `TipcSrvDestroy` tries to send the value of `Server_Disconnect_Mode` to RTserver in a `DISCONNECT` message. This allows an RTclient to easily change `Server_Disconnect_Mode` before disconnecting (otherwise RTclient would have to disconnect, change the option, reconnect, and then disconnect again).

The `DISCONNECT` message is not sent if `TipcSrvDestroy` is being called as the result of a nonrecoverable error on the connection to RTserver. When a nonrecoverable error occurs, no more messages can be sent or received.

`Server_Disconnect_Mode` and the `DISCONNECT` message type are somewhat related to warm connections in RTclient, but they control how RTserver handles GMD for a disconnecting RTclient. This provides more robustness for GMD in RTserver, as RTserver does not care if RTclient crashes or exits cleanly; all disconnects are treated the same.

GMD Area

The internal full pathnames used for the RTclient file-based GMD area are simpler than they are for peer-to-peer connections, because the connection to RTserver has a fixed purpose. See the reference page for `TipcSrvGmdFileCreate` in the *TIBCO SmartSockets Application Programming Interface* reference for full details on RTclient GMD area pathnames.

RTclient GMD area holds one highest sequence number for each sending RTclient it has received a message (sent with GMD) from. This enables each receiving RTclient to differentiate the sequence numbers from all sending RTclients. The sending RTclients are identified by the value of their `Unique_Subject` option.

With peer-to-peer connections, messages must be resent with GMD by calling the function `TipcConnGmdResend`; RTclient does this resending automatically. Both `TipcSrvCreate` and `TipcSrvDestroy(T_IPC_SRV_CONN_WARM)` call `TipcSrvGmdResend` to resend messages from the GMD area. With a warm connection to RTserver, this is needed to prime the warm connection with old GMD messages so as to preserve the proper outgoing message order. It should almost never be necessary to call `TipcSrvGmdResend` explicitly; it is provided for completeness only.

This automatic resending helps RTclients, especially SmartSockets Modules such as RTie, to use GMD more easily without requiring any extra coding. The only disadvantage of the automatic use of `TipcSrvGmdResend` is that if `TipcSrvGmdFileDelete` is used, it must be called before calling `TipcSrvCreate`.

The `TipcSrvCreate` function does not create the file-based GMD area directories until they are needed. This helps to prevent large numbers of unused directories from being created for large publish-subscribe projects.

File-based GMD and Connections to Multiple RTservers

For normal global connections, you must set the `Unique_Subject` option to a value other than the default to use file-based GMD. For multiple connections that connect to multiple RTservers from a single RTclient, you must also set a unique subject for each connection, but not using the `Unique_Subject` option. Instead, you set a connection's unique subject using named options or by specifying the unique subject as an argument when you create the connection. Connections are allowed to share the same unique subject only if their projects are different or if they connect to different RTserver clouds.

When you specify file-based GMD, a sub-directory is created in which the messages are written, to be stored in case they need to be resent due to a failure. By default, this sub-directory is named after the RTclient's unique subject. If you have connections that share the same unique subject, you must specify different GMD sub-directories for each of them, or else they both attempt to use the same GMD sub-directory and cause file conflicts.

To specify a GMD sub-directory for a connection, set the named option `Server_Gmd_Dir_Name` for the connection or call the `TipcSrvConnSetGmdDirName` function.

RTserver GMD Considerations

How GMD Works in RTserver

RTserver does not keep any GMD information in files. The publishing RTclient has full responsibility for maintaining a persistent copy of the message. This speeds up and simplifies GMD, as only the publishing process performs the majority of disk file operations. If RTserver crashes and restarts, the publishing RTclient can resend the messages from its GMD area.

When an RTclient publishes the first message using GMD to a subject, RTserver starts collecting subject subscription information from the appropriate other RTservers to accurately track GMD accounting. This increases the scalability of publish-subscribe GMD due to the fact that only the relevant RTservers dynamically exchange GMD information. The RTclient API function `TipcSrvSubjectGmdInit` can also be used to manually initialize GMD accounting for a subject to which messages will be published. The RTserver option `Gmd_Publish_Timeout` specifies the amount of time RTserver continues to maintain GMD accounting for a subject that has not been recently published to with GMD; when this timeout is reached, the inter-RTserver GMD information is no longer accumulated.

RTserver does keep GMD information in memory, although not the same way as a memory-only GMD area. When a direct RTclient sends a message with GMD, the RTserver creates a record containing:

- the message itself is saved so it can be resent easily (by incrementing the message reference count, so that no extra memory management is needed)
- the list of all RTclients (both direct and indirect) that are currently subscribing to the destination subject; this is the list of expected receivers
- an empty list of the RTclients that have acknowledged delivery successfully
- an empty list of the RTclients for which GMD has failed

RTserver then routes the message normally by sending a copy of the message to all appropriate direct RTclients and RTservers. As the subscribing RTclients receive, process, and acknowledge the message, they send `GMD_ACK` messages, which are routed back to the original RTserver.

When the original RTserver receives an acknowledgment, it removes the receiving RTclient from the list of expected receivers and adds it to the list of successes. If failures occur, such as an expected receiver RTclient, whose `Server_Disconnect_Mode` option is set to `gmd_failure`, disconnects without acknowledging the message, the failed RTclient is moved from expected list to the

failed list, and a GMD_NACK message is sent to the original RTclient publisher. The publishing RTclient receives the GMD_NACK message, then builds and processes a GMD_FAILURE message to notify the program of an asynchronous GMD failure.

By keeping the original GMD message in memory, RTserver can easily and quickly resend the message to reconnecting warm RTclients without having to notify or wait for the original publishing RTclient. This speeds up recovery and also ensures that the reconnecting warm RTclient receives the old messages first before receiving any new messages. RTserver also uses the list of expected receivers to prevent resends from going to RTclients that start subscribing to the subject after the original message was sent.

If RTserver receives a GMD_DELETE message, it destroys the in-memory GMD record about the message, effectively terminating GMD for the message (or at least terminating resending of the message). If RTserver receives a GMD_STATUS_CALL message, it constructs and returns a GMD_STATUS_RESULT message containing the lists of successful, failed, and pending subscribers.

When RTserver receives enough acknowledgment messages (one in the case of T_IPC_DELIVERY_SOME and all of them in the case of T_IPC_DELIVERY_ALL), RTserver destroys the in-memory GMD record about the message and sends a GMD_ACK message to the original RTclient that sent the message. This simplifies GMD for the publisher, which does not have to track multiple acknowledgments.

Configuring RTserver for GMD

To configure an RTserver for GMD, set the GMD-related options in the RTserver startup command file or using a CONTROL message. If you want to use the default configuration, you do not need to reset the options from their default values. These are the GMD options for RTserver:

Client_Reconnect_Timeout	specifies the maximum amount of time, in seconds, to allow a warm RTclient to reconnect. The default value is 30 seconds.
Gmd_Publish_Timeout	specifies the amount of time, in seconds, to maintain GMD accounting information after the last publish with GMD as a subject. The default value is 300 seconds.
Zero_Recv_Gmd_Failure	specifies how guaranteed message delivery should complete when there are no RTclients subscribing to the destination subject of the message (that is, the subject membership list is empty).

For more information on these options, see Chapter 8, Options Reference.

Combining GMD and Monitoring

Guaranteed message delivery and monitoring are both powerful features in the SmartSockets publish-subscribe architecture, and when combined, they enable the easy development of robust, fault-tolerant programs. When using GMD, monitoring is especially useful to help make decisions when recovering from GMD failures.

The only restriction when using GMD with monitoring is that the delivery modes `T_IPC_DELIVERY_SOME` and `T_IPC_DELIVERY_ALL` should not be used with the `MON_*` message types. `RTserver` handles all the `MON_*` message types internally, and does not support routing acknowledgments for these message types.

Handling GMD Failures with RTclients and RTservers

As discussed in Handling GMD Failures on page 341, recovery from GMD_FAILURE messages is very application-specific, and thus SmartSockets cannot provide a generic solution on its own. For example, when processing a GMD_FAILURE message, the sender could cancel a transaction by sending another message to all the RTclients that did receive the message. RTclient recovery can be more complicated because each message sent is routed to all RTclients subscribing to the destination subject. However, through the use of the monitoring capabilities described in Chapter 5, Project Monitoring, a sending RTclient can easily probe the state of the project and make recovery decisions based on the monitoring information. In any case, it is essential that the reason for the failure be ascertained by either the application or the administrator before the message is re-sent. Re-sending the message before the cause of the failure is fixed will only compound the failure.

In addition to the GMD_FAILURE connection process callback `TipCbConnProcessGmdFailure` that is created for all connections, RTclient's connection to RTserver by default has a process callback for GMD_FAILURE message that uses `TipCbSrvProcessGmdFailure` as the callback function. `TipCbSrvProcessGmdFailure` calls `TipcSrvGmdMsgServerDelete` to terminate GMD in RTserver for the failed message.

`TipCbSrvProcessGmdFailure` is intended as a sample callback that is designed only to reclaim memory in RTserver when GMD fails. More sophisticated programs should create their own process callbacks for GMD_FAILURE messages to perform actions such as user-defined recovery procedures. See the reference page for `TipCbSrvProcessGmdFailure` in the *TIBCO SmartSockets Application Programming Interface* reference for full details on this callback.

From a GMD_FAILURE process callback, the function `TipcSrvGmdMsgServerDelete` can be used to notify RTserver to terminate GMD for a specific message. For example:

```
if (!TipcSrvGmdServerDelete(msg)) {
    /* error */
}
```

From a GMD_FAILURE process callback, the function `TipcSrvGmdMsgStatus` can be used to query RTserver for the GMD status of a specific message. For example:

```
if (!TipcSrvGmdMsgStatus(msg)) {
    /* error */
}

/* wait up to 10 seconds for RTserver to respond */
mt = TipcMtLookupByNum(T_MT_GMD_STATUS_RESULT);
if (mt == NULL) {
    /* error */
}
status_msg = TipcSrvMsgSearchType(10.0, mt);
if (status_msg == NULL) {
    /* error */
}
/* access fields from msg and make recovery decisions */
```


Chapter 5 **Project Monitoring**

The publish-subscribe architecture of SmartSockets allows RTclient processes to easily send messages to each other, independent of where they reside on the network. In addition to enabling easy development of distributed applications, RTserver, RTmon, and RTclient processes also have monitoring capabilities and naming services. These additional services help you to examine detailed information about your project and determine where processes are located in your network.

Topics

- *Monitoring Overview, page 360*
- *Monitoring Composition, page 361*
- *Polling, page 382*
- *Watching, page 408*
- *Advanced Monitoring, page 426*

Monitoring Overview

From within an RTclient, hundreds of pieces of information can be gathered in real time about all parts of a running SmartSockets project:

- RTclients — names, extension data, message buffers, message traffic statistics, message types, options, CPU usage, memory usage, node, options, current time, subscribed subjects, and so on
- RTservers — names, message buffers, message traffic statistics, connections, options, CPU usage, memory usage, node, options, current time, and so on
- Subjects — names, RTclients that are subscribing, and so on
- Projects — names

This information can be polled once to provide a one-time snapshot of information or watched to provide asynchronous updates when changes occur. A monitoring request can specify either a specific object or all objects matching a wildcard scope filter. The information is delivered to the requesting program in standard message types, prefixed with T_MT_MON_ (for brevity, the T_MT_ portion is omitted from this point on). The fields of these monitoring message types contain the information the RTclient is interested in.

Monitoring is built into SmartSockets, requiring no user-defined code. Just as RTserver and RTclient are layered on top of the function of connections, monitoring is layered on top of the RTserver and RTclient architecture. There is no monitoring available for peer-to-peer connections or for programs that do not use the SmartSockets API or C++ Class Library.

This chapter describes SmartSockets process monitoring, watching and polling for information, and using the monitoring API, TipcMon*. The focus of this chapter is on RTclients and RTservers using global connections. In rare cases, an RTclient might need to use multiple connections to RTservers, using special multiple connections instead of a single global connection. Monitoring this type of connection requires the use of the TipcSrvMon* API instead of TipcMon*. Although the monitoring tasks and concepts are similar to those described in this chapter, an entirely different set of APIs is used. For more information on this type of monitoring, see the *TIBCO SmartSockets API Quick Reference* and the *TIBCO SmartSockets Application Programming Interface* reference.

Monitoring Composition

All monitoring of a SmartSockets project goes through RTserver. Typically, a request for particular information is sent to RTserver, then RTserver proceeds in one of these ways:

- RTserver accesses the information in its internal tables and sends it back to the requesting RTclient.
- RTserver forwards the request to another RTclient or RTserver, which retrieves the information and sends it back to the first RTserver (to be delivered back to the requesting RTclient).
- RTserver notifies RTclient or RTserver to send the information whenever it changes. When it does change, RTserver receives the information and sends it to the RTclient that made the original request.

A request is initiated either:

- through the TipcMon* API functions, designed for monitoring the global connection between an RTclient and RTserver

The TipcMon* functions can be used in conjunction with any of the other Tipc* API functions that work with the global connection. They make it simple for an RTclient to monitor a project, as well as exchange messages with other RTclients.

- through the TipcSrvMon* API functions, designed for monitoring special multiple connections between an RTclient and RTserver

The TipcSrvMon* functions can be used in conjunction with any of the other Tipc* API functions designed to work with multiple connections, such as the TipcSrvConn* APIs. For more information on using the TipcSrvMon* APIs to monitor multiple connection projects, see the *TIBCO SmartSockets API Quick Reference* and the *TIBCO SmartSockets Application Programming Interface* reference.

- through an RTmon command

RTmon is a standard RTclient for monitoring that provides both a visual point-and-click interface and access to monitoring information through a command interface. Using RTmon is described in Chapter 6, Using RTmon, on page 445.

Issuing a command in RTmon or calling a monitoring API function results in a message, specifying the request, being sent to RTserver. These standard messages for requesting information should never be constructed. You should always use an API function or RTmon command to initiate a monitoring message.

To initiate a monitoring request in an RTclient, call one of the functions in the TipcMon* API. This sends a message of type MON_*_SET_WATCH (if watching) or MON_*_POLL_CALL (if polling) to RTserver. The information is returned to the program by an RTserver using a MON_*_STATUS or MON_*_POLL_RESULT message. When the information returns, an RTclient message process callback (created with TipcSrvProcessCbCreate) can be used to process the message. Complete examples of polling and watching for information are shown in the sections Polling on page 382 and Watching on page 408.

Where Monitoring Information Resides

The requested information resides in either an RTserver or an RTclient. Information that resides in an RTserver, to which your client is directly connected, is expected to be returned quickly. These are examples of the information that reside in an RTserver:

- project names
- subjects in existence, and which RTclients are subscribed to them
- RTserver message buffer-related information
- generic RTserver process information (such as CPU time or memory usage)
- nodes on which all processes are running
- names and values of the options in RTserver
- connections between RTservers

These are examples of information that resides in an RTclient:

- extension data, which is data created by an RTclient
- RTclient message buffer-related information
- generic RTclient process information (such as CPU time or memory usage)
- names and values of the options in RTclient
- message types RTclient has defined

Information that resides in an RTclient or an indirect RTserver may or may not come back quickly, depending on how busy the RTclient or the RTserver is. If many incoming messages are buffered for an RTclient, the monitoring request message is put on the queue of RTclient in priority order and is processed just like any other message. If the queue is long, it can be some time before the request is serviced and the results returned to the requesting program. The best way to ensure that monitoring requests are serviced quickly is to assign high priorities to the monitoring message types.

By default, all monitoring message types are initialized to the message type priority (if set) or to the value of the option `Default_Msg_Priority` (if the message type priority is unknown). If nothing is done to change the priority of a monitoring message type, all monitoring requests are issued with priority zero (the default value of the `Default_Msg_Priority` option in the RTclient issuing the request). This can be changed using the `TipcMtSetPriority` API call, as described in Priority on page 18, to change the priority of the `MON_*_POLL_CALL` and `MON_*_SET_WATCH` message types. In a similar manner, the information returned can be given higher priority by changing the priority on the `MON_*_POLL_RESULT` and `MON_*_STATUS` message types, which are described in detail later in this chapter.

Specifying Items to be Monitored

In `TipcMon*Poll` or `TipcMon*SetWatch` functions, common parameters are the names of the RTclient, RTserver, subject, message type, or option to monitor. The parameter can be the name of a specific type of object to monitor, a wildcarded subject name to select multiple objects (for RTclient and RTserver names only), or the constant `T_IPC_MON_ALL`. `T_IPC_MON_ALL` is used to specify all objects of a certain type or all objects matching the wildcarded value in the option `Monitor_Scope` (see the next section for details). In addition, the `TipcMon*` functions with no parameters (such as `TipcMonServerNamesPoll`) also can be thought of as having an implicit parameter, which is always `T_IPC_MON_ALL`.

The name parameter should be specified as a string (`T_STR`) and follow these rules:

- when specifying a *client_name*, use either:
 - the unique subject of the RTclient to monitor
 - a wildcarded subject name to match many RTclients
 - `T_IPC_MON_ALL` to monitor all RTclients in the project matching `Monitor_Scope`

Here are examples:

```
/* EXAMPLE 1: Get general information on the RTclient
   identified by the unique subject "my_program" */
if (!TipcMonClientGeneralPoll("/my_program")) ...

/* EXAMPLE 2: Get general information on all RTclients */
if (!TipcMonClientGeneralPoll("/...")) ...
```

- when specifying a *server_name*, use either:
 - the unique subject of the RTserver to monitor
 - a wildcarded subject name to match many RTservers
 - T_IPC_MON_ALL to monitor all RTservers in the RTserver group matching Monitor_Scope
- when specifying a *subject_name*, use the name of the subject to monitor or T_IPC_MON_ALL to monitor all subjects in the project matching Monitor_Scope
- when specifying a *msg_type*, use the name of the message type to monitor or T_IPC_MON_ALL to monitor all message types created in the RTclient
- when specifying an *option_name*, use the name of the option to monitor or T_IPC_MON_ALL to monitor all options in RTclient or RTserver

When a wildcarded value or the T_IPC_MON_ALL value is used, the request is sent to all matching RTclients and RTservers that hold the information. In this case, there are multiple responses and an RTclient message process callback should be used to process the results.

If a poll is initiated on an item that does not exist (for example, no subject exists that matches *subject_name*), the poll fails and no message is returned. If a watch is initiated on an item that does not exist, RTserver or RTclient saves the request and services it once the item becomes available. In both polls and watches, it is dangerous to block indefinitely and wait for the result of a poll or watch to come back. Always give a finite timeout when waiting for monitoring results to return.

Monitor Scope and T_IPC_MON_ALL

The `Monitor_Scope` option specifies the level of interest for SmartSockets monitoring in those monitoring categories with no parameters (such as RTclient names poll) or an RTclient or RTserver parameter of `T_IPC_MON_ALL` (such as RTclient time watch). `Monitor_Scope` acts as a filter that prevents a large project from overloading a monitoring program. The default is `"/**"`, which matches all subject names at the first level of the hierarchical subject namespace. All monitoring information is enabled (all filtering is disabled) if `Monitor_Scope` is set to `"/..."`, which matches all names. The value `"/..."` should be used with caution on large projects, as this gathers monitoring information for the entire project, and may cause noticeable performance degradation.

RTclient and RTserver monitoring parameters allow a wildcard subject name to be used for matching because the unique subject of an RTserver or RTclient cannot be a wildcarded value. Subject monitoring parameters do not allow a wildcard subject name to be used for matching because subject names can use wildcards, and thus wildcard subject names in monitoring requests are treated literally and are not expanded to all matching subjects. For example, a monitoring request to watch all subscribers to the subject `"/stocks/..."` is ambiguous (that is, does the monitoring request want only the subscribers for the wildcard subject `"/stocks/..."` or the subscribers for all subjects matching `"/stocks/..."`?). There is a way to get a matching list for subject monitoring through the use of the `T_IPC_MON_ALL` value, in which case all subjects matching the value of the option `Monitor_Scope` are used.

The `Monitor_Scope` option is not used for option names and message type names, but these two types instead use `T_IPC_MON_ALL` to indicate that all objects should be monitored. `T_IPC_MON_ALL` has two meanings:

- for subject names, RTclient names, and RTserver names, `T_IPC_MON_ALL` means to use the value of the `Monitor_Scope` option
- for all other names, such as option names and message type names, `T_IPC_MON_ALL` means to use all values

The `Monitor_Scope` option makes it easy to write programs with `T_IPC_MON_ALL` to monitor a certain portion of the hierarchical subject namespace. Simply by changing `Monitor_Scope` (for example, from `"/stocks/computers/..."` to `"/stocks/auto/..."`), the entire `T_IPC_MON_ALL`-using program can change its monitoring focus without any code changes.

Watching or Polling: When to Use

As described earlier, information is either polled or watched. Polling should be used in these situations:

- Collect information that cannot be watched (such as an RTclient's message types or RTclient extension data).
- Collect information that the program must have before continuing. An example of this is to ensure that the needed RTclients are running in a project before continuing.
- Collect information that only needs to be monitored occasionally or at regular intervals, not every time it changes (for example, if you only want to see the names of all subjects once for each minute).

Watching, on the other hand, should be used in these situations:

- Collect information that cannot be polled (for example, the messages that an RTclient is receiving).
- Collect information that needs to be looked at whenever it changes (for example, when an RTclient subscribes or unsubscribes to a subject).
- Collect information that may or may not be available. If you do not know that the RTclient that holds the information of interest is available, watching should be used as RTserver maintains the watch information and primes the RTclient with the proper settings when it becomes available.

Of course, watching and polling can be used together within the same RTclient to gather information as needed.

Monitoring Message Types

The core of SmartSockets monitoring is the message types. The message types can be thought of as describing the monitoring protocol function. These message types are accessible through various bindings: a C API, a C++ API, and the RTmon command interface.

These message types fall into one of these general categories:

- MON_*_POLL_CALL — initiates a poll request
- MON_*_POLL_RESULT — holds results of a poll request
- MON_*_SET_WATCH — initiates a watch request
- MON_*_STATUS — holds results of a watch request

The table lists the monitoring message types and their associated grammar. The T_MT_ prefix is omitted from the message types for the sake of brevity.

Message Type (T_MT_ is omitted)	Grammar
MON_CLIENT_BUFFER_POLL_CALL	str /*client_name*/
MON_CLIENT_BUFFER_POLL_RESULT	str /*client_name*/ int4 /*msg_queue_count*/ int4 /*msg_queue_byte_count*/ int4 /*read_buffer_count*/ int4 /*write_buffer_count*/
MON_CLIENT_BUFFER_SET_WATCH	str /*client_name*/ bool /*watch_status*/
MON_CLIENT_BUFFER_STATUS	str /*client_name*/ int4 /*msg_queue_count*/ int4 /*msg_queue_byte_count*/ int4 /*read_buffer_count*/ int4 /*write_buffer_count*/
MON_CLIENT_CB_POLL_CALL	str /*client_name*/

Message Type (T_MT_ is omitted)	Grammar
MON_CLIENT_CB_POLL_RESULT	<div>str /*client_name*/</div> <div>int4 /*num_global_read_cb*/</div> <div>int4 /*num_global_write_cb*/</div> <div>int4 /*num_global_process_cb*/</div> <div>int4 /*num_global_queue_cb*/</div> <div>int4 /*num_default_cb*/</div> <div>int4 /*num_error_cb*/</div> <div>int4 /*num_server_create_cb*/</div> <div>int4 /*num_server_destroy_cb*/</div>
MON_CLIENT_CONGESTION_SET_WATCH	<div>str /*client_name*/</div> <div>int4 /*high_water*/</div> <div>int4 /*low_water*/</div> <div>bool /*watch_status*/</div>
MON_CLIENT_CONGESTION_STATUS	<div>int4 /*size*/</div> <div>int4 /*threshold*/</div> <div>str /*client_name*/</div> <div>bool /*high_water*/</div>
MON_CLIENT_CPU_POLL_CALL	<div>str /*client_name*/</div>
MON_CLIENT_CPU_POLL_RESULT	<div>str /*client_name*/</div> <div>real4 /*cpu_utilization*/</div>
MON_CLIENT_EXT_POLL_CALL	<div>str /*client_name*/</div>
MON_CLIENT_EXT_POLL_RESULT	<div>str /*client_name*/</div> <div>int4 /*number of named fields*/</div> <div>{named_field}</div>
MON_CLIENT_GENERAL_POLL_CALL	<div>str /*client_name*/</div>

Message Type (T_MT_ is omitted)	Grammar
MON_CLIENT_GENERAL_POLL_RESULT	str /*client_name*/ str /*ident*/ str /*node_name*/ str /*user_name*/ int4 /*pid*/ str /*project*/ str /*server_name*/ str /*arch*/ int4 /*current_sbrk*/ int4 /*sbrk_delta_since_startup*/ int2 /*int_format*/ int2 /*real_format*/ str /*logical_conn_name_to_server*/ int4_array /*counted_licenses*/ str_array /*extra_licenses*/ str_array /*subject_subscribe*/
MON_CLIENT_INFO_POLL_CALL	str /*client_name*/

Message Type (T_MT_ is omitted)	Grammar
MON_CLIENT_INFO_POLL_RESULT	<div>str /*client_name*/</div> <div>str /*ident*/</div> <div>str /*node_name*/</div> <div>str /*user_name*/</div> <div>int4 /*pid*/</div> <div>str /*project*/</div> <div>str /*server_name*/</div> <div>str /*arch*/</div> <div>int4 /*current_sbrk*/</div> <div>int4 /*sbrk_delta_since_startup*/</div> <div>int2 /*int_format*/</div> <div>int2 /*real_format*/</div> <div>str /*logical_conn_name_to_server*/</div> <div>int4 /*num_subscribes*/</div> <div>real4 /*cpu_utilization*/</div>
MON_CLIENT_MSG_RECV_SET_WATCH	<div>str /*client_name*/</div> <div>str /*msg_type_name*/</div> <div>bool /*watch_status*/</div>
MON_CLIENT_MSG_RECV_STATUS	<div>str /*client_name*/</div> <div>msg /*recv_msg*/</div> <div>bool /*insert_flag*/</div> <div>int4 /*queue_pos*/</div> <div>binary /*msg_id*/</div> <div>binary /*queue_msg_id_array*/</div>
MON_CLIENT_MSG_SEND_SET_WATCH	<div>str /*client_name*/</div> <div>str /*msg_type_name*/</div> <div>bool /*watch_status*/</div>

Message Type (T_MT_ is omitted)	Grammar
MON_CLIENT_MSG_SEND_STATUS	str /*client_name*/ msg /*send_msg*/
MON_CLIENT_MSG_TRAFFIC_POLL_CALL	str /*client_name*/
MON_CLIENT_MSG_TRAFFIC_POLL_RESULT	str /*client_name*/ int4 /*total_msg_send*/ int4 /*total_msg_rcv*/ int4 /*total_byte_send*/ int4 /*total_byte_rcv*/ int8 /*total_msg_send_8*/ int8 /*total_msg_rcv_8*/ int8 /*total_byte_send_8*/ int8 /*total_byte_rcv_8*/
MON_CLIENT_MSG_TYPE_EX_POLL_CALL	str /*client_name*/ str /*msg_type_name*/

Message Type (T_MT_ is omitted)	Grammar
MON_CLIENT_MSG_TYPE_EX_POLL_RESULT	<div>str /*client_name*/</div> <div>str_array /*msg_type_name*/</div> <div>int4_array /*num*/</div> <div>str_array /*grammar*/</div> <div>int4_array /*priority_known*/</div> <div>int2_array /*priority*/</div> <div>int4_array /*delivery_mode*/</div> <div>int4_array /*user_prop*/</div> <div>int4_array /*num_read_cb*/</div> <div>int4_array /*num_write_cb*/</div> <div>int4_array /*num_process_cb*/</div> <div>int4_array /*num_queue_cb*/</div> <div>int8_array /*total_msg_send*/</div> <div>int8_array /*total_msg_recv*/</div> <div>int8_array /*total_byte_send*/</div> <div>int8_array /*total_byte_recv*/</div>
MON_CLIENT_MSG_TYPE_POLL_CALL	<div>str /*client_name*/</div> <div>str /*msg_type_name*/</div>

Message Type (T_MT_ is omitted)	Grammar
MON_CLIENT_MSG_TYPE_POLL_RESULT	str /*client_name*/ {str /*msg_type_name*/ int4 /*num*/ str /*grammar*/ int4 /*priority_known*/ int2 /*priority*/ int4 /*delivery_mode*/ int4 /* user_prop*/ int4 /*num_read_cb*/ int4 /*num_write_cb*/ int4 /*num_process_cb*/ int4 /*num_queue_cb*/ int4 /*total_msg_send*/ int4 /*total_msg_rcv*/ int4 /*total_byte_send*/ int4 /*total_byte_rcv*/}
MON_CLIENT_NAMES_NUM_POLL_CALL	/*empty*/
MON_CLIENT_NAMES_NUM_POLL_RESULT	int4 /*num_clients*/
MON_CLIENT_NAMES_POLL_CALL	/*empty*/
MON_CLIENT_NAMES_POLL_RESULT	str_array /*client_names*/ str_array /*client_info_strs*/ str_array /*server_names*/
MON_CLIENT_NAMES_SET_WATCH	bool /*watch_status*/

Message Type (T_MT_ is omitted)	Grammar
MON_CLIENT_NAMES_STATUS	str_array /*client_names*/ str_array /*client_info_strs*/ str_array /*server_names*/ str /*created_client*/ str /*destroyed_client*/ str /*reason_for_disconnect*/
MON_CLIENT_OPTION_POLL_CALL	str /*client_name*/ str /*option_name*/
MON_CLIENT_OPTION_POLL_RESULT	str /*client_name*/ {str /*option_name*/ int2 /*type*/ str /*value_str*/ bool /*required*/ str_array /*legal_vals*/}
MON_CLIENT_SUBJECT_EX_POLL_CALL	str /*client_name*/ str /*subject_name*/
MON_CLIENT_SUBJECT_EX_POLL_RESULT	str /*client_name*/ str_array /*subject_name*/ int8_array /*total_msg_send*/ int8_array /*total_msg_rcv*/ int8_array /*total_byte_send*/ int8_array /*total_byte_rcv*/
MON_CLIENT_SUBJECT_POLL_CALL	str /*client_name*/ str /*subject_name*/

Message Type (T_MT_ is omitted)	Grammar
MON_CLIENT_SUBJECT_POLL_RESULT	str /*client_name*/ {str /*subject_name*/ int4 /*total_msg_send*/ int4 /*total_msg_rcv*/ int4 /*total_byte_send*/ int4 /*total_byte_rcv*/}
MON_CLIENT_SUBSCRIBE_NUM_POLL_CALL	str /*client_name*/
MON_CLIENT_SUBSCRIBE_NUM_POLL_RESULT	str_array /*client_names*/ int4_array /*num_subscribes*/
MON_CLIENT_SUBSCRIBE_POLL_CALL	str /*client_name*/
MON_CLIENT_SUBSCRIBE_POLL_RESULT	{str /*client_name*/ str_array /*subscribe_subject_names*/}
MON_CLIENT_SUBSCRIBE_SET_WATCH	str /*client_name*/ bool /*watch_status*/
MON_CLIENT_SUBSCRIBE_STATUS	str /*client_name*/ str_array /*subscribe_subject_names*/ str /*start_subject*/ str /*stop_subject*/
MON_CLIENT_TIME_POLL_CALL	str /*client_name*/
MON_CLIENT_TIME_POLL_RESULT	str /*client_name*/ real8 /*current_time*/ str /*current_time_str*/ real8 /*wall_time*/ str /*wall_time_str*/ real8 /*cpu_time*/ int4 /*frame_count*/

Message Type (T_MT_ is omitted)	Grammar
MON_CLIENT_TIME_SET_WATCH	str /*client_name*/ bool /*watch_status*/
MON_CLIENT_TIME_STATUS	str /*client_name*/ real8 /*current_time*/ str /*current_time_str*/ real8 /*wall_time*/ str /*wall_time_str*/ real8 /*cpu_time*/ int4 /*frame_count*/
MON_CLIENT_VERSION_POLL_CALL	str /*client_name*/
MON_CLIENT_VERSION_POLL_RESULT	str /*client_name*/ int4 /*version*/
MON_PROJECT_NAMES_POLL_CALL	/*empty*/
MON_PROJECT_NAMES_POLL_RESULT	str_array /*project_names*/
MON_PROJECT_NAMES_SET_WATCH	bool /*watch_status*/
MON_PROJECT_NAMES_STATUS	str_array /*project_names*/ str /*created_project*/ str /*destroyed_project*/
MON_SERVER_BUFFER_POLL_CALL	str /*server_name*/ str /*connected_process_name*/
MON_SERVER_BUFFER_POLL_RESULT	str /*server_name*/ {str /*connected_process_name*/ int4 /*msg_queue_count*/ int4 /*msg_queue_byte_count*/ int4 /*read_buffer_count*/ int4 /*write_buffer_count*/}

Message Type (T_MT_ is omitted)	Grammar
MON_SERVER_CONGESTION_SET_WATCH	str /*server_name*/ str /*connected_process_name*/ int4 /*high_water*/ int4 /*low_water*/ bool /*watch_status*/
MON_SERVER_CONGESTION_STATUS	int4 /*size*/ int4 /*threshold*/ str /*server_name*/ str /*connected_process_name*/ bool /*high_water*/
MON_SERVER_CONN_POLL_CALL	/*empty*/
MON_SERVER_CONN_POLL_RESULT	str_array /*server1_names*/ str_array /*server2_names*/ str_array /*conn_names*/ int4_array /*weights*/
MON_SERVER_CONN_SET_WATCH	bool /*watch_status*/
MON_SERVER_CONN_STATUS	str_array /*server1_names*/ str_array /*server2_names*/ str_array /*conn_names*/ int4_array /*weights*/ str /*start_server1*/ str /*start_server2*/ str /*stop_server1*/ str /*stop_server2*/
MON_SERVER_CPU_POLL_CALL	str /*server_name*/
MON_SERVER_CPU_POLL_RESULT	str /*server_name*/ real4 /*cpu_utilization*/

Message Type (T_MT_ is omitted)	Grammar
MON_SERVER_GENERAL_POLL_CALL	str /*server_name*/
MON_SERVER_GENERAL_POLL_RESULT	str /*server_name*/ str /*ident*/ str /*node_name*/ str /*user_name*/ int4 /*pid*/ str /*arch*/ int4 /*current_sbrk*/ int4 /*sbrk_delta_since_startup*/ int2 /*int_format*/ int2 /*real_format*/ str /*cmd_file_name*/ bool /*no_daemon_flag*/ bool /*cmd_mode_flag*/ str_array /*direct_client_names*/ str_array /*direct_server_names*/ str_array /*server_subscribes*/ str_array /*client_subscribes*/
MON_SERVER_MAX_CLIENT_LICENSES_SET_WATCH	str /*server_name*/ bool /*watch_status*/
MON_SERVER_MAX_CLIENT_LICENSES_STATUS	int4 /*max_licenses*/ str /*client_name*/ str /*server_name*/
MON_SERVER_MSG_TRAFFIC_EX_POLL_CALL	str /*server_name*/ str /*connected_process_name*/

Message Type (T_MT_ is omitted)	Grammar
MON_SERVER_MSG_TRAFFIC_EX_POLL_RESULT	str /*server_name*/ str_array /*connected_process_name*/ int8_array /*total_msg_send*/ int8_array /*total_msg_rcv*/ int8_array /*total_byte_send*/ int8_array /*total_byte_rcv*/
MON_SERVER_MSG_TRAFFIC_POLL_CALL	str /*server_name*/ str /*connected_process_name*/
MON_SERVER_MSG_TRAFFIC_POLL_RESULT	str /*server_name*/ { str /*connected_process_name*/ int4 /*total_msg_send*/ int4 /*total_msg_rcv*/ int4 /*total_byte_send*/ int4 /*total_byte_rcv*/ }
MON_SERVER_NAMES_POLL_CALL	/*empty*/
MON_SERVER_NAMES_POLL_RESULT	str_array /*server_names*/ str_array /*server_info_strs*/
MON_SERVER_NAMES_SET_WATCH	bool /*watch_status*/
MON_SERVER_NAMES_STATUS	str_array /*server_names*/ str_array /*server_info_strs*/ str /*created_server*/ str /*destroyed_server*/
MON_SERVER_OPTION_POLL_CALL	str /*server_name*/ str /*option_name*/

Message Type (T_MT_ is omitted)	Grammar
MON_SERVER_OPTION_POLL_RESULT	str /*server_name*/ {str /*option_name*/ int2 /*type*/ str /*value_str*/ bool /*required*/ str_array /*legal_vals*/}
MON_SERVER_ROUTE_POLL_CALL	str /*orig_server_name*/ str /*dest_server_name*/
MON_SERVER_ROUTE_POLL_RESULT	str /*orig_server_name*/ {str /*dest_server_name*/ str_array /*connected_server_names*/ int4 /*distance*/ str_array /*route_to_server*/}
MON_SERVER_START_TIME_POLL_CALL	str /*server_name*/
MON_SERVER_START_TIME_POLL_RESULT	str /*server_name*/ real8 /*start_time*/ real8 /*elapsed_time*/
MON_SERVER_TIME_POLL_CALL	str /*server_name*/
MON_SERVER_TIME_POLL_RESULT	str /*server_name*/ real8 /*wall_time*/ str /*wall_time_str*/ real8 /*cpu_time*/
MON_SERVER_VERSION_POLL_CALL	str /*server_name*/
MON_SERVER_VERSION_POLL_RESULT	str /*server_name*/ int4 /*version*/
MON_SUBJECT_NAMES_POLL_CALL	/*empty*/

Message Type (T_MT_ is omitted)	Grammar
MON_SUBJECT_NAMES_POLL_RESULT	str_array /*subject_names*/
MON_SUBJECT_NAMES_SET_WATCH	bool /*watch_status*/
MON_SUBJECT_NAMES_STATUS	str_array /*subject_names*/ str /*created_subject*/ str /*destroyed_subject*/
MON_SUBJECT_SUBSCRIBE_POLL_CALL	str /*subject_name*/
MON_SUBJECT_SUBSCRIBE_POLL_RESULT	{str /*subject_name*/ str_array /*subscribe_client_names*/}
MON_SUBJECT_SUBSCRIBE_SET_WATCH	str /*subject_name*/ bool /*watch_status*/
MON_SUBJECT_SUBSCRIBE_STATUS	str /*subject_name*/ str_array /*subscribe_client_names*/ str /*start_client*/ str /*stop_client*/

Polling

Polling for information allows you to receive a one-time snapshot of information. Polling can be done at regular time intervals by issuing poll requests repeatedly. All polling of a project goes through RTserver. Typically, a request for particular information is sent to RTserver, then RTserver retrieves the information from its internal tables, from other RTservers, or from RTclients and sends it back to the requesting program.

The type of information polled about projects, RTservers, RTclients, and subjects, is shown in the table. The table lists the entity from where the information is available (RTclient, RTserver, subject), a brief description of the information available, whether the information can be watched as well as polled, and the final column shows the TipcMon* API call that initiates the polling and then the monitoring message type returned. The prefix T_MT_ is omitted from the message type for the sake of brevity.

Entity	Information Available	Watch?	API Function and Message Type Returned (T_MT_ omitted)
Project	Names	Yes	TipcMonProjectNamesPoll MON_PROJECT_NAMES_POLL_RESULT
RTclient	Read and write message buffers	Yes	TipcMonClientBufferPoll MON_CLIENT_BUFFER_POLL_RESULT
RTclient	Callback summary	No	TipcMonClientCbPoll MON_CLIENT_CB_POLL_RESULT
RTclient	CPU percentage used	No	TipcMonClientCpuPoll MON_CLIENT_CPU_POLL_RESULT
RTclient	Extension data	No	TipcMonClientExtPoll MON_CLIENT_EXT_POLL_RESULT
RTclient	General (process)	No	TipcMonClientGeneralPoll MON_CLIENT_GENERAL_POLL_RESULT
RTclient	General information	No	TipcMonClientInfoPoll MON_CLIENT_INFO_POLL_RESULT

Entity	Information Available	Watch?	API Function and Message Type Returned (T_MT_ omitted)
RTclient	Message traffic	No	TipcMonClientMsgTrafficPoll MON_CLIENT_MSG_TRAFFIC_POLL_RESULT
RTclient	Message types	No	TipcMonClientMsgTypeExPoll MON_CLIENT_MSG_TYPE_EX_POLL_RESULT
RTclient	Names	Yes	TipcMonClientNamesPoll MON_CLIENT_NAMES_POLL_RESULT
RTclient	Options	No	TipcMonClientOptionPoll MON_CLIENT_OPTION_POLL_RESULT
RTclient	Summary of messages sent or received from a subject	No	TipcMonClientSubjectExPoll MON_CLIENT_SUBJECT_EX_POLL_RESULT
RTclient	Subjects subscribing to	Yes	TipcMonClientSubscribePoll MON_CLIENT_SUBSCRIBE_POLL_RESULT
RTclient	Wall time, CPU time used	Yes	TipcMonClientTimePoll MON_CLIENT_TIME_POLL_RESULT
RTclient	Version	No	TipcMonClientVersionPoll MON_CLIENT_VERSION_POLL_RESULT
RTserver	Number of RTclients in a project	No	TipcMonClientNamesNumPoll MON_CLIENT_NAMES_NUM_POLL_RESULT
RTserver	Number of subjects a client subscribes to	No	TipcMonClientSubscribeNumPoll MON_CLIENT_SUBSCRIBE_NUM_POLL_RESULT
RTserver	Read and write message buffers	No	TipcMonServerBufferPoll MON_SERVER_BUFFER_POLL_RESULT
RTserver	Connections	Yes	TipcMonServerConnPoll MON_SERVER_CONN_POLL_RESULT

Entity	Information Available	Watch?	API Function and Message Type Returned (T_MT_ omitted)
RTserver	CPU percentage used	No	TipcMonServerCpuPoll MON_SERVER_CPU_POLL_RESULT
RTserver	General (process)	No	TipcMonServerGeneralPoll MON_SERVER_GENERAL_POLL_RESULT
RTserver	Message traffic	No	TipcMonServerMsgTrafficExPoll MON_SERVER_MSG_TRAFFIC_EX_POLL_RESULT
RTserver	Names	Yes	TipcMonServerNamesPoll MON_SERVER_NAMES_POLL_RESULT
RTserver	Options	No	TipcMonServerOptionPoll MON_SERVER_OPTION_POLL_RESULT
RTserver	Routes	No	TipcMonServerRoutePoll MON_SERVER_ROUTE_POLL_RESULT
RTserver	Start time and elapsed time	No	TipcMonServerStartTimePoll MON_SERVER_START_TIME_POLL_RESULT
RTserver	Wall time, CPU time used	No	TipcMonServerTimePoll MON_SERVER_TIME_POLL_RESULT
RTserver	Version	No	TipcMonServerVersionPoll MON_SERVER_VERSION_POLL_RESULT
subject	Names	Yes	TipcMonSubjectNamesPoll MON_SUBJECT_NAMES_POLL_RESULT
subject	Clients subscribing to	Yes	TipcMonSubjectSubscribePoll MON_SUBJECT_SUBSCRIBE_POLL_RESULT

Some information that can be polled can also be watched. See Watching on page 408 for more details on watching information.

To initiate a polling request in your program, call an API function, `TipcMonTypePoll`, where *Type* is the kind of information you wish to request. For example, a call to `TipcMonSubjectSubscribePoll` initiates a poll for information about which RTclients are subscribing to a specified subject, and a call to `TipcMonClientTimePoll` initiates a poll for time information about an RTclient.

Calling a `TipcMonTypePoll` function causes a message of type `MON_TYPE_POLL_CALL` to be sent to RTserver, where *TYPE* is the kind of information being requested. This *TYPE* is very similar to the *Type* used in the name of the API function as shown in these examples:

- `TipcMonSubjectSubscribePoll` sends a message of type `MON_SUBJECT_SUBSCRIBE_POLL_CALL`
- `TipcMonClientTimePoll` sends a message of type `MON_CLIENT_TIME_POLL_CALL`

RTserver may already hold the information requested, RTserver may need to ask other RTservers for the information, or RTserver may need to forward the message to one or more RTclients to retrieve the information. If RTserver holds the information, the response should come back quickly. For example, you can poll for the names of all projects that RTserver knows about.

If the RTclient has the information, the response may or may not come back quickly depending on how busy the RTclient is. For example, you can poll for information about an RTclient's message buffers. If RTserver must gather the requested information from an RTclient, the RTclient must read and process messages through calls to `TipcSrvMainLoop`, `TipcSrvMsgNext`, `TipcMsgSearch`, or `TipcMsgSearchType` in order for the poll request to be returned.

RTserver returns requested information in a `MON_TYPE_POLL_RESULT` message, where *TYPE* matches the *TYPE* in the `MON_TYPE_POLL_CALL` that initiated the request. The message types are in pairs of `MON_TYPE_POLL_CALL` and `MON_TYPE_POLL_RESULT` as shown in these examples:

- `TipcMonSubjectSubscribePoll` sends a message of type `MON_SUBJECT_SUBSCRIBE_POLL_CALL`, and a message of type `MON_SUBJECT_SUBSCRIBE_POLL_RESULT` is returned by RTserver.
- `TipcMonClientTimePoll` sends a message of type `MON_CLIENT_TIME_POLL_CALL`, and a message of type `MON_CLIENT_TIME_POLL_RESULT` is returned by RTserver.

A complete listing and description of these message types are described in Polling Message Types on page 389. The message grammars for all monitoring message types are described in Monitoring Message Types on page 367.

Processing Poll Results

When the results of a poll are returned in a `MON_TYPE_POLL_RESULT` message, the message is processed in one of these two ways:

- blocking and using the API function `TipcSrvMsgSearch` to search for the message containing the poll results
- using an `RTclient` message process callback (created with `TipcSrvProcessCbCreate`)

Typically, after performing a one-time poll for information, the program blocks and waits for the result before continuing. For example, you can poll to find out if any `RTclients` are subscribing to a subject prior to sending a message. When polling often, or watching for information, using callbacks is usually the preferred method, as it is unclear when the monitoring results may be returned to the program. The sections that follow present examples of how to process poll results using both methods.

Using the `TipcMsgSearchType` Function

This example shows how to issue a poll request and then use `TipcMsgSearchType` to block and search for the results (for up to 10 seconds):

```
T_IPC_MT mt;
T_IPC_MSG msg;
T_STR *project_names;
T_INT4 num_project_names;
/* send the poll request out to RTserver */
if (!TipcMonProjectNamesPoll()) {
    /* error */
}

mt = TipcMtLookupByNum(T_MT_MON_PROJECT_NAMES_POLL_RESULT);
if (mt == NULL) {
    /* error */
}

/* wait up to 10 seconds for the poll result */
msg = TipcSrvMsgSearchType(10.0, mt);
if (msg == NULL) {
    /* error */
}
```

```
/* access the fields of the returned message */
if (!TipcMsgSetCurrent(msg, 0)) {
    /* error */
}

if (!TipcMsgNextStrArray(msg, &project_names, &num_project_names))
{
    /* error */
}

/* do whatever here with the poll results */

/* clean up */
if (!TipcMsgDestroy(msg)) {
    /* error */
}
```

Using Callbacks

This example shows how to use an RTclient message process callback (created with TipcSrvProcessCbCreate) to process the message containing the poll result:

```

/*..process_project_names_poll - callback to process
   MON_PROJECT_NAMES_POLL_RESULT */
static void T_ENTRY process_project_names_poll(
    T_IPC_CONN conn,
    T_IPC_CONN_PROCESS_CB_DATA data,
    T_CB_ARG arg)
{
    T_STR *project_names;
    T_INT4 num_project_names;

    /* access the fields of the returned message */
    if (!TipcMsgSetCurrent(data->msg, 0)) {
        /* error */
    }

    if (!TipcMsgNextStrArray(data->msg, &project_names,
        &num_project_names)) {
        /* error */
    }

    /* do whatever here with poll results */
} /* process_project_names_poll */

/* ===== */
/* ... later, in main program */

T_IPC_MT mt;

/* create callback to process MON_PROJECT_NAMES poll results */
mt = TipcMtLookupByNum(T_MT_MON_PROJECT_NAMES_POLL_RESULT);
if (mt == NULL) {
    /* error */
}

if (TipcSrvProcessCbCreate(mt, process_project_names_poll, NULL)
    == NULL){
    /* error */
}

/* send the poll request out to RTserver */
if (!TipcMonProjectNamesPoll()) {
    /* error */
}

/* TipcSrvMainLoop may be called here */

```

Polling Message Types

This section contains a complete listing of each polling message type. For each message pair (MON_TYPE_POLL_CALL and MON_TYPE_POLL_RESULT), a description of the type of information, the API function, and where the information is collected from is shown. Information gathered from RTserver should come back quickly, while information gathered from RTclient may or may not.

The T_MT_ prefix is omitted from the name of each message type for the sake of brevity.

MON_CLIENT_BUFFER_POLL_*

The MON_CLIENT_BUFFER_POLL_CALL message type is used to request message-related buffer information about one or more RTclients in a project. The polled RTclients respond by sending back MON_CLIENT_BUFFER_POLL_RESULT messages.

API Function: TipcMonClientBufferPoll

Message Initiated: MON_CLIENT_BUFFER_POLL_CALL

Message Returned: MON_CLIENT_BUFFER_POLL_RESULT

Info Gathered From: RTclient

MON_CLIENT_CB_POLL_*

The MON_CLIENT_CB_POLL_CALL message type is used to request callback information about one or more RTclients in a project. The polled RTclients respond by sending back a MON_CLIENT_CB_POLL_RESULT message.

API Function: TipcMonClientCbPoll

Message Initiated: MON_CLIENT_CB_POLL_CALL

Message Returned: MON_CLIENT_CB_POLL_RESULT

Info Gathered From: RTclient

MON_CLIENT_CPU_POLL_*

The MON_CLIENT_CPU_POLL_CALL message type is used to request CPU usage information about one or more RTclients in a project. The polled RTclients respond by sending back MON_CLIENT_CPU_POLL_RESULT messages.

API Function: TipcMonClientCpuPoll

Message Initiated: MON_CLIENT_CPU_POLL_CALL

Message Returned: MON_CLIENT_CPU_POLL_RESULT

Info Gathered From: RTclient

MON_CLIENT_EXT_POLL_*

The MON_CLIENT_EXT_POLL_CALL message type is used to request RTclient extension data. Extension data is information created by an RTclient. You must use the TipcMonExt* APIs to define and update the fields of an RTclient's MON_CLIENT_EXT_POLL_RESULT message for extension data. The polled RTclients respond by sending back MON_CLIENT_EXT_POLL_RESULT messages.

API Function: TipcMonClientExtPoll

Message Initiated: MON_CLIENT_EXT_POLL_CALL

Message Returned: MON_CLIENT_EXT_POLL_RESULT

Info Gathered From: RTclient

MON_CLIENT_GENERAL_POLL_*

The MON_CLIENT_GENERAL_POLL_CALL message type is used to request general information about one or more RTclients in a project. The polled RTclients respond by sending back a MON_CLIENT_GENERAL_POLL_RESULT message.

API Function: TipcMonClientGeneralPoll

Message Initiated: MON_CLIENT_GENERAL_POLL_CALL

Message Returned: MON_CLIENT_GENERAL_POLL_RESULT

Info Gathered From: RTclient

MON_CLIENT_INFO_POLL_*

The MON_CLIENT_INFO_POLL_CALL message type is used to request general information, including CPU usage, about one or more RTclients in a project. The polled RTclients respond by sending back MON_CLIENT_INFO_POLL_RESULT messages.

API Function: TipcMonClientInfoPoll

Message Initiated: MON_CLIENT_INFO_POLL_CALL

Message Returned: MON_CLIENT_INFO_POLL_RESULT

Info Gathered From: RTclient

MON_CLIENT_MSG_TRAFFIC_POLL_*

The MON_CLIENT_MSG_TRAFFIC_POLL_CALL message type is used to request message traffic information about one or more RTclients in a project. The polled RTclients respond by sending back a MON_CLIENT_MSG_TRAFFIC_POLL_RESULT message.

API Function: TipcMonClientMsgTrafficPoll

Message Initiated: MON_CLIENT_MSG_TRAFFIC_POLL_CALL

Message Returned: MON_CLIENT_MSG_TRAFFIC_POLL_RESULT

Info Gathered From: RTclient

MON_CLIENT_MSG_TYPE_EX_POLL_*

The MON_CLIENT_MSG_TYPE_EX_POLL_CALL message type is used to request message type information about one or more RTclients in a project. The polled RTclients respond by sending back a MON_CLIENT_MSG_TYPE_POLL_EX_RESULT message.

API Function: TipcMonClientMsgTypeExPoll

Message Initiated: MON_CLIENT_MSG_TYPE_EX_POLL_CALL

Message Returned: MON_CLIENT_MSG_TYPE_EX_POLL_RESULT

Info Gathered From: RTclient



This poll supersedes the MON_CLIENT_MSG_TYPE_POLL_* poll for new development. See TipcMonClientMsgTypePoll in the *TIBCO SmartSockets Application Programming Interface* guide for more information. The MON_CLIENT_MSG_TYPE_POLL_* is retained for compatibility with clients prior to release 6.7.

MON_CLIENT_NAMES_NUM_POLL_*

The MON_CLIENT_NAMES_NUM_POLL_CALL message type is used to request the number of RTclients in an RTserver cloud. The polled RTclients respond by sending back MON_CLIENT_NAMES_NUM_POLL_RESULT messages.

API Function: TipcMonClientNamesNumPoll

Message Initiated: MON_CLIENT_NAMES_NUM_POLL_CALL

Message Returned: MON_CLIENT_NAMES_NUM_POLL_RESULT

Info Gathered From: RTserver

MON_CLIENT_NAMES_POLL_*

The MON_CLIENT_NAMES_POLL_CALL message type is used to request the running RTclient names in the current project. RTserver responds by sending back a MON_CLIENT_NAMES_POLL_RESULT message.

API Function: TipcMonClientNamesPoll

Message Initiated: MON_CLIENT_NAMES_POLL_CALL

Message Returned: MON_CLIENT_NAMES_POLL_RESULT

Info Gathered From: RTserver

MON_CLIENT_OPTION_POLL_*

The MON_CLIENT_OPTION_POLL_CALL message type is used to request option information about one or more RTclients in a project. The polled RTclients respond by sending back a MON_CLIENT_OPTION_POLL_RESULT message.

API Function: TipcMonClientOptionPoll

Message Initiated: MON_CLIENT_OPTION_POLL_CALL

Message Returned: MON_CLIENT_OPTION_POLL_RESULT

Info Gathered From: RTclient

MON_CLIENT_SUBJECT_EX_POLL_*

The MON_CLIENT_SUBJECT_EX_POLL_CALL message type is used to request subject information about one or more RTclients in a project. The polled RTclients respond by sending back a MON_CLIENT_SUBJECT_EX_POLL_RESULT message.

API Function: TipcMonClientSubjectExPoll

Message Initiated: MON_CLIENT_SUBJECT_EX_POLL_CALL

Message Returned: MON_CLIENT_SUBJECT_EX_POLL_RESULT

Info Gathered From: RTclient



This poll supersedes the MON_CLIENT_SUBJECT_POLL_* poll for new development. See TipcMonClientSubjectPoll in the *TIBCO SmartSockets Application Programming Interface* guide for more information. The MON_CLIENT_SUBJECT_POLL_* is retained for compatibility with clients prior to release 6.7.

MON_CLIENT_SUBSCRIBE_NUM_POLL_*

The MON_CLIENT_SUBSCRIBE_NUM_POLL_CALL message type is used to request the number of subjects subscribed to by one or more RTclients in a project. The polled RTclients respond by sending back MON_CLIENT_SUBSCRIBE_NUM_POLL_RESULT messages.

API Function: TipcMonClientSubscribeNumPoll

Message Initiated: MON_CLIENT_SUBSCRIBE_NUM_POLL_CALL

Message Returned: MON_CLIENT_SUBSCRIBE_NUM_POLL_RESULT

Info Gathered From: RTserver

MON_CLIENT_SUBSCRIBE_POLL_*

The MON_CLIENT_SUBSCRIBE_POLL_CALL message type is used to request the current subjects to which one or more RTclients in a project are subscribing. RTserver responds by sending back a MON_CLIENT_SUBSCRIBE_POLL_RESULT message for each RTclient specified.

API Function: TipcMonClientSubscribePoll

Message Initiated: MON_CLIENT_SUBSCRIBE_POLL_CALL

Message Returned: MON_CLIENT_SUBSCRIBE_POLL_RESULT

Info Gathered From: RTserver

MON_CLIENT_TIME_POLL_*

The MON_CLIENT_TIME_POLL_CALL message type is used to request time information about one or more RTclients in a project. The polled RTclients respond by sending back MON_CLIENT_TIME_POLL_RESULT messages.

API Function: TipcMonClientTimePoll

Message Initiated: MON_CLIENT_TIME_POLL_CALL

Message Returned: MON_CLIENT_TIME_POLL_RESULT

Info Gathered From: RTclient

MON_CLIENT_VERSION_POLL_*

The MON_CLIENT_VERSION_POLL_CALL message type is used to request client version of one or more RTclients in a project. The polled RTclients respond by sending back MON_CLIENT_VERSION_POLL_RESULT messages.

API Function: TipcMonClientVersionPoll

Message Initiated: MON_CLIENT_VERSION_POLL_CALL

Message Returned: MON_CLIENT_VERSION_POLL_RESULT

Info Gathered From: RTclient

MON_PROJECT_NAMES_POLL_*

The MON_PROJECT_NAMES_POLL_CALL message type is used to request the current project names from RTserver. RTserver responds by sending back a MON_PROJECT_NAMES_POLL_RESULT message.

API Function: TipcMonProjectNamesPoll

Message Initiated: MON_PROJECT_NAMES_POLL_CALL

Message Returned: MON_PROJECT_NAMES_POLL_RESULT

Info Gathered From: RTserver

MON_SERVER_BUFFER_POLL_*

The MON_SERVER_BUFFER_POLL_CALL message type is used to request message-related buffer information about one or more RTservers in an RTserver group. The polled RTservers respond by sending back a MON_SERVER_BUFFER_POLL_RESULT message.

API Function: TipcMonServerBufferPoll

Message Initiated: MON_SERVER_BUFFER_POLL_CALL

Message Returned: MON_SERVER_BUFFER_POLL_RESULT

Info Gathered From: RTserver

MON_SERVER_CONN_POLL_*

The MON_SERVER_CONN_POLL_CALL message type is used to request connection information about one or more RTservers in an RTserver group. The polled RTservers respond by sending back a MON_SERVER_CONN_POLL_RESULT message.

API Function: TipcMonServerConnPoll

Message Initiated: MON_SERVER_CONN_POLL_CALL

Message Returned: MON_SERVER_CONN_POLL_RESULT

Info Gathered From: RTserver

MON_SERVER_CPU_POLL_*

The MON_SERVER_CPU_POLL_CALL message type is used to request CPU usage information about one or more RTservers in a project. The polled RTclients respond by sending back MON_SERVER_CPU_POLL_RESULT messages.

API Function: TipcMonServerCpuPoll

Message Initiated: MON_SERVER_CPU_POLL_CALL

Message Returned: MON_SERVER_CPU_POLL_RESULT

Info Gathered From: RTserver

MON_SERVER_GENERAL_POLL_*

The MON_SERVER_GENERAL_POLL_CALL message type is used to request general process information about one or more RTservers in an RTserver group. The polled RTservers respond by sending back a MON_SERVER_GENERAL_POLL_RESULT message.

API Function: TipcMonServerGeneralPoll

Message Initiated: MON_SERVER_GENERAL_POLL_CALL

Message Returned: MON_SERVER_GENERAL_POLL_RESULT

Info Gathered From: RTserver

MON_SERVER_MSG_TRAFFIC_EX_POLL_*

The MON_SERVER_MSG_TRAFFIC_EX_POLL_CALL message type is used to request message traffic information about one or more RTservers in an RTserver group. The polled RTservers respond by sending back a MON_SERVER_MSG_TRAFFIC_EX_POLL_RESULT message.

API Function: TipcMonServerMsgTrafficExPoll

Message Initiated: MON_SERVER_MSG_TRAFFIC_EX_POLL_CALL

Message Returned: MON_SERVER_MSG_TRAFFIC_EX_POLL_RESULT

Info Gathered From: RTserver



This poll supersedes the MON_SERVER_MSG_TRAFFIC_POLL_* poll for new development. See TipcMonServerMsgTrafficPoll in the *TIBCO SmartSockets Application Programming Interface* guide for more information. The MON_SERVER_MSG_TRAFFIC_POLL_* is retained for compatibility with servers prior to release 6.7.

MON_SERVER_NAMES_POLL_*

The MON_SERVER_NAMES_POLL_CALL message type is used to request the running RTserver names. RTserver responds by sending back a MON_SERVER_NAMES_POLL_RESULT message.

API Function: TipcMonServerNamesPoll

Message Initiated: MON_SERVER_NAMES_POLL_CALL

Message Returned: MON_SERVER_NAMES_POLL_RESULT

Info Gathered From: RTserver

MON_SERVER_OPTION_POLL_*

The MON_SERVER_OPTION_POLL_CALL message type is used to request option information about one or more RTservers in an RTserver group. The polled RTservers respond by sending back a MON_SERVER_OPTION_POLL_RESULT message.

API Function: TipcMonServerOptionPoll

Message Initiated: MON_SERVER_OPTION_POLL_CALL

Message Returned: MON_SERVER_OPTION_POLL_RESULT

Info Gathered From: RTserver

MON_SERVER_ROUTE_POLL_*

The MON_SERVER_ROUTE_POLL_CALL message type is used to request route information about one or more RTservers in an RTserver group. The polled RTservers respond by sending back a MON_SERVER_ROUTE_POLL_RESULT message.

API Function: TipcMonServerRoutePoll

Message Initiated: MON_SERVER_ROUTE_POLL_CALL

Message Returned: MON_SERVER_ROUTE_POLL_RESULT

Info Gathered From: RTserver

MON_SERVER_START_TIME_POLL_*

The MON_SERVER_START_TIME_POLL_CALL message type is used to request start time and elapsed time information about one or more RTservers in an RTserver group. The polled RTservers respond by sending back MON_SERVER_START_TIME_POLL_RESULT messages.

API Function: TipcMonStartServerTimePoll

Message Initiated: MON_SERVER_START_TIME_POLL_CALL

Message Returned: MON_SERVER_START_TIME_POLL_RESULT

Info Gathered From: RTserver

MON_SERVER_TIME_POLL_*

The MON_SERVER_TIME_POLL_CALL message type is used to request time information (wall time and CPU time used) about one or more RTservers in an RTserver group. The polled RTservers respond by sending back a MON_SERVER_TIME_POLL_RESULT message.

API Function: TipcMonServerTimePoll

Message Initiated: MON_SERVER_TIME_POLL_CALL

Message Returned: MON_SERVER_TIME_POLL_RESULT

Info Gathered From: RTserver

MON_SERVER_VERSION_POLL_*

The MON_SERVER_VERSION_POLL_CALL message type is used to request server version of one or more RTservers in a project. The polled RTservers respond by sending back MON_SERVER_VERSION_POLL_RESULT messages.

API Function: TipcMonServerVersionPoll

Message Initiated: MON_SERVER_VERSION_POLL_CALL

Message Returned: MON_SERVER_VERSION_POLL_RESULT

Info Gathered From: RTserver

MON_SUBJECT_NAMES_POLL_*

The MON_SUBJECT_NAMES_POLL_CALL message type is used to request the current subject names in the current project. RTserver responds by sending back a MON_SUBJECT_NAMES_POLL_RESULT message.

API Function: TipcMonSubjectNamesPoll

Message Initiated: MON_SUBJECT_NAMES_POLL_CALL

Message Returned: MON_SUBJECT_NAMES_POLL_RESULT

Info Gathered From: RTserver

MON_SUBJECT_SUBSCRIBE_POLL_*

The MON_SUBJECT_SUBSCRIBE_POLL_CALL message type is used to request the current RTclients that are subscribing to a specified subject or all subjects. RTserver responds by sending back a MON_SUBJECT_SUBSCRIBE_POLL_RESULT message for each subject.

API Function: TipcMonSubjectSubscribePoll

Message Initiated: MON_SUBJECT_SUBSCRIBE_POLL_CALL

Message Returned: MON_SUBJECT_SUBSCRIBE_POLL_RESULT

Info Gathered From: RTserver

Polling Example

The following is a complete C source example of a program that continues to poll for message traffic information at a regular interval, where the interval (in seconds) is one of the command line arguments passed in to the program.

The source code files for this example are located in these directories:

UNIX:

\$RTHOME/examples/smrtsock/manual

OpenVMS:

RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]

Windows:

%RTHOME%\examples\smrtsock\manual

The online source files have additional #ifdefs to provide C++ support. These #ifdefs are not shown to simplify the example.

/*

polltraf.c

USAGE

polltraf <project> <unique_subject> <poll_interval>

This program gathers message traffic information on the RTclient specified in the command line arg <unique_subject> in project <project>. The info is polled for on a regular interval as specified in <poll_interval>. You can gather message traffic info on all clients in a project by specifying "/"... for <unique_subject>. <poll_interval> is always specified in seconds (as an integer).

Sample usage:

polltraf tank my_ie 10

polltraf tank /... 30

*/

#include <rtworks/ipc.h>

```

/*
=====
*/
/*..cb_default - default callback */
static void T_ENTRY cb_default
(
    T_IPC_CONN conn,
    T_IPC_CONN_DEFAULT_CB_DATA data,
    T_CB_ARG arg
)
{
    if (!TipcMsgPrintError(data->msg)) {
        TutOut("Could not print message: error <%s>.\n",
            TutErrStrGet());
    }
} /* cb_default */

/* =====
*/
/*..process_traffic_poll - callback to process a
                        MON_CLIENT_MSG_TRAFFIC_POLL_RESULT message
*/
static void T_ENTRY process_traffic_poll
(
    T_IPC_CONN conn,
    T_IPC_CONN_PROCESS_CB_DATA data,
    T_CB_ARG arg
)
{
    T_IPC_MSG msg = data->msg;
    T_STR client_name;
    T_INT4 total_msg_rcv_4;
    T_INT4 total_msg_send_4;
    T_INT4 total_byte_rcv_4;
    T_INT4 total_byte_send_4;
    T_INT8 total_msg_rcv_8;
    T_INT8 total_msg_send_8;
    T_INT8 total_byte_rcv_8;
    T_INT8 total_byte_send_8;

    /* Set current field */
    if (!TipcMsgSetCurrent(msg, 0)) {
        TutOut("Could not set current field of message: error <%s>.\n",
            TutErrStrGet());
        return;
    }
}

```

```

/* Get the fields from the message */
if (!TipcMsgRead(data->msg, T_IPC_FT_STR, &client_name,
                    T_IPC_FT_INT4, &total_msg_rcv_4,
                    T_IPC_FT_INT4, &total_msg_snd_4,
                    T_IPC_FT_INT4, &total_byte_rcv_4,
                    T_IPC_FT_INT4, &total_byte_snd_4,
                    NULL)) {
    TutOut("Unable to access message traffic message: error
<%s>.\n",
        TutErrStrGet());
    return;
}

TutOut("Summary of message traffic for RTclient <%s>\n",
client_name);
if (TipcMsgRead(data->msg, T_IPC_FT_INT8, &total_msg_rcv_8,
                    T_IPC_FT_INT8, &total_msg_snd_8,
                    T_IPC_FT_INT8, &total_byte_rcv_8,
                    T_IPC_FT_INT8, &total_byte_snd_8,
                    NULL)) {
    /* Print out the new information received */
    TutOut("    (Received 64-bit data)\n");
    TutOut("    Total messages received = " T_INT8_SPEC "\n",
        total_msg_rcv_8);
    TutOut("    Total messages sent      = " T_INT8_SPEC "\n",
        total_msg_snd_8);
    TutOut("    Total bytes received     = " T_INT8_SPEC "\n",
        total_byte_rcv_8);
    TutOut("    Total bytes sent        = " T_INT8_SPEC "\n",
        total_byte_snd_8);
}
else {
    /* Print out the new information received */
    TutOut("    (Received 32-bit data)\n");
    TutOut("    Total messages received = " T_INT4_SPEC "\n",
        total_msg_rcv_4);
    TutOut("    Total messages sent      = " T_INT4_SPEC "\n",
        total_msg_snd_4);
    TutOut("    Total bytes received     = " T_INT4_SPEC "\n",
        total_byte_rcv_4);
    TutOut("    Total bytes sent        = " T_INT4_SPEC "\n",
        total_byte_snd_4);
}

TutOut("=====\n");
;

} /* process_traffic_poll */

```

```

/* =====
*/
int main
(
    int argc,
    char **argv
)
{
    T_STR project_name;
    T_STR unique_subject;
    T_REAL8 poll_interval;
    T_IPC_MT mt;
    T_OPTION option;

    /* Check the command line arguments */
    if (argc != 4) {
        TutOut("Usage: polltraf <project> <unique_subject> ");
        TutOut("<poll_interval>\n");
        TutExit(T_EXIT_FAILURE);
    }

    /* Save pointers to command line arguments */
    project_name = argv[1];
    unique_subject = argv[2];
    poll_interval = atof(argv[3]);

    /* Check that the polling interval is greater than zero */
    if (poll_interval <= 0.0) {
        TutOut("Poll_Interval must be greater than zero.\n");
        TutExit(T_EXIT_FAILURE);
    }

    TutOut("Polling RTclient <%s> in project <%s> ",
        unique_subject, project_name);
    TutOut("every <%g> seconds...\n", poll_interval);

    /* Set the project name */
    option = TutOptionLookup("project");
    if (option == NULL) {
        TutOut("Could not look up option <project>: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    if (!TutOptionSetEnum(option, project_name)) {
        TutOut("Could not set option <project>: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    /* Connect to RTserver */
    if (!TipcSrvCreate(T_IPC_SRV_CONN_FULL)) {
        TutOut("Could not connect to RTserver: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
}

```

```

/* Create callback to process CLIENT_MSG_TRAFFIC poll results */
mt = TipcMtLookupByNum(T_MT_MON_CLIENT_MSG_TRAFFIC_POLL_RESULT);
if (mt == NULL) {
    TutOut("Could not look up message type ");
    TutOut("MON_CLIENT_MSG_TRAFFIC_POLL_RESULT: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

if (TipcSrvProcessCbCreate(mt, process_traffic_poll, NULL) ==
NULL){
    TutOut("Could not set up callback to process ");
    TutOut("MON_CLIENT_MSG_POLL_RESULT: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* Create default callback to receive unwanted message types */
if (TipcSrvDefaultCbCreate(cb_default, NULL) == NULL) {
    TutOut("Could not create default callback: error <%s>.\n",
        TutErrStrGet());
}

/* Read and process poll results every "poll_interval" seconds */
while (1) {
    TutOut("==> Polling RTclient <%s> for message traffic
info...\n",
        unique_subject);

    /* Initiate poll for message traffic info about an RTclient */
    if (!TipcMonClientMsgTrafficPoll(unique_subject)) {
        TutOut("Unable to poll for message traffic for RTclient
<%s>.\n",
            unique_subject);
        TutOut(" error <%s>.\n", TutErrStrGet());
    }

    /* Process the poll results which arrive */
    if (!TipcSrvMainLoop(poll_interval)) {
        TutOut("TipcSrvMainLoop failed: error <%s>.\n",
            TutErrStrGet());
    }
}

/* This line should not be reached. */
return T_EXIT_FAILURE;
} /* main */

```

This program retrieves both input and output information about the RTclients' message traffic. The poll for traffic information is initiated by the call to `TipcMonClientMsgTrafficPoll`. Note that the argument to this function is *unique_subject*. This argument is passed in as a command line argument and must be either:

- the unique subject of a specified RTclient
- a wildcarded subject name, which indicates the request should be sent to all RTclients with a unique subject matching the wildcarded subject name
- an at sign (@), which indicates the request should be sent to all RTclients that match `Monitor_Scope`

In this program, the poll is sent to the specified RTclient or many RTclients at a regular interval, which is controlled by the call to `TipcSrvMainLoop`. The `poll_interval` timeout causes `TipcSrvMainLoop` to read and process all incoming messages for that many seconds before giving up control.

The call to `TipcMonClientMsgTrafficPoll` results in one or a series (one for each RTclient polled) of `MON_CLIENT_MSG_TRAFFIC_POLL_RESULT` messages being returned to the program from RTserver. An RTclient message process callback, `process_traffic_poll`, is created to process messages of that type.

The callback function `process_traffic_poll` accesses the fields of the poll result message and prints the requested information. If no RTclient exists that matches the *unique_subject* passed to `TipcMonClientMsgTrafficPoll`, no results are printed.

Compiling, Linking, and Running

To compile, link, and run the `polltraf` program, first you must either copy the program to your own directory or have write permission to the directory:

UNIX:

```
$RTHOME/examples/smrtsock/manual
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]
```

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

You are also going to run a program with some message traffic.

Step 1 Copy the receive.c program from this directory

Note that this program is part of the `ss_tutorial` project and subscribes to the `lesson5` subject.

UNIX:

```
$RTHOME/examples/smrtsock/tutorial/lesson5
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.TUTORIAL.LESSON5]
```

Windows:

```
%RTHOME%\examples\smrtsock\tutorial\lesson5
```

Step 2 Compile and link the polltraf and receive programs**UNIX:**

```
$ rtlink polltraf.c -o polltraf.x
$ rtlink receive.c -o receive.x
```

OpenVMS:

```
$ cc polltraf.c
$ rtlink /exec=polltraf.exe polltraf.obj
$ cc receive.c
$ rtlink /exec=receive.exe receive.obj
```

Windows:

```
$ nmake /f pollw32m.mak
$ nmake /f recw32m.mak
```

To run the programs, start the `receive` process in one terminal emulator window and then the `polltraf` process in another terminal emulator window.

Step 3 Start the receive program in the first window**UNIX:**

```
$ receive.x
```

OpenVMS:

```
$ run receive.exe
```

Windows:

```
$ receive.exe
```


Step 4 Start the polltraf program in the second window**UNIX:**

```
$ polltraf.x ss_tutorial /... 10
```

OpenVMS:

```
$ polltraf := $sys$disk:[ ]polltraf.exe
$ polltraf ss_tutorial /... 10
```

Windows:

```
$ polltraf.exe ss_tutorial /... 10
```

The name `/...` indicates that all RTclients should be polled. Here is an example of the output displayed:

```
Polling RTclient </...> in project <ss_tutorial> every <10>
seconds...
```

```
Connecting to project <ss_tutorial> on <_node> RTserver.
```

```
Using local protocol.
```

```
Message from RTserver: Connection established.
```

```
Start subscribing to subject </_grissom_1037>.
```

```
==> Polling RTclient </...> for message traffic info...
```

```
Summary of message traffic for RTclient </_grissom_1028>
```

```
Total messages received = 19
Total messages sent      = 19
Total bytes received     = 792
Total bytes sent         = 792
```

```
=====
Summary of message traffic for RTclient </_grissom_1037>
```

```
Total messages received = 1
Total messages sent      = 2
Total bytes received     = 64
Total bytes sent         = 64
```

```
=====
==> Polling RTclient </...> for message traffic info...
```

```
Summary of message traffic for RTclient </_grissom_1028>
```

```
Total messages received = 20
Total messages sent      = 20
Total bytes received     = 904
Total bytes sent         = 904
```

```
=====
Summary of message traffic for RTclient </_grissom_1037>
```

```
Total messages received = 3
Total messages sent      = 4
Total bytes received     = 240
Total bytes sent         = 240
```

```
=====
```

Notice that when a poll takes place, two results are reported. One is for the receive program, the second is for the polltraf program. When `/...` is used to specify all RTclients, polltraf even receives information about itself (which initiated the poll).

Watching

Watching information is very different from polling. When watching information, the RTclient that initiated the watch is notified asynchronously, through a message, whenever the item(s) of interest changes. All watching of a project goes through RTserver. Typically, a request to watch particular information is sent to RTserver. RTserver then notifies the process holding the information (the same RTserver, a different RTserver, or an RTclient) that a program is interested in watching the specified information. Then, whenever the information changes, the process holding the information sends it back to the requesting program in a message. Unlike polling, which is a one time request for information, watching causes status messages to continue to be sent until watching is turned off for that particular information.

The type of information watched about projects, RTservers, RTclients, and subjects, is shown in the table. The table lists the entity from which the information is available (such as RTclient, RTserver, or subject), a brief description of the information that is available, whether the information can be polled as well as watched, and the final column shows the TipcMon* API call that initiates the watching and the monitoring message type returned whenever the information of interest changes. The prefix T_MT_ is omitted from the message type for the sake of brevity.

Entity	Information Available	Poll?	API Function and Message Type Returned (T_MT_ omitted)
Project	Names	Yes	TipcMonProjectNamesSetWatch MON_PROJECT_NAMES_STATUS
RTclient	Read and write message buffers	Yes	TipcMonClientBufferSetWatch MON_CLIENT_BUFFER_STATUS
RTclient	Write buffer congestion	No	TipcMonClientCongestionSetWatch MON_CLIENT_CONGESTION_SET_WATCH
RTclient	Messages being received	No	TipcMonClientMsgRecvSetWatch MON_CLIENT_MSG_RECV_STATUS
RTclient	Messages being sent out	No	TipcMonClientMsgSendSetWatch MON_CLIENT_MSG_RECV_STATUS

Entity	Information Available	Poll?	API Function and Message Type Returned (T_MT_ omitted)
RTclient	Subjects subscribing to	Yes	TipcMonClientSubscribeSetWatch MON_CLIENT_SUBSCRIBE_STATUS
RTclient	Wall time, CPU time used	Yes	TipcMonClientTimeSetWatch MON_CLIENT_TIME_STATUS
RTclient	Names	Yes	TipcMonClientNamesSetWatch MON_CLIENT_NAMES_STATUS
RTserver	Write buffer congestion	No	TipcMonServerCongestionSetWatch MON_SERVER_CONGESTION_SET_WATCH
RTserver	Connections	Yes	TipcMonServerConnSetWatch MON_SERVER_CONN_STATUS
RTserver	Licenses	No	TipcMonServerMaxClientLicensesSetWatch MON_SERVER_MAX_CLIENT_LICENSES_SET_WATCH
RTserver	Names	Yes	TipcMonServerNamesSetWatch MON_SERVER_NAMES_STATUS
subject	Names	Yes	TipcMonSubjectNamesSetWatch MON_SUBJECT_NAMES_STATUS
subject	Clients subscribing to	Yes	TipcMonSubjectSubscribeSetWatch MON_SUBJECT_SUBSCRIBE_STATUS

Most of the information that can be watched can also be polled. See Polling on page 382 for more details on polling for information.

To set up watching in your program, you call an API function, `TipcMonTypeSetWatch`, with the `watch_status` parameter set to `TRUE`, where *Type* is the kind of information to watch. For example, a call to `TipcMonSubjectSubscribeSetWatch(subject_name, TRUE)` turns on watching for information about which RTclients are subscribing to a specified subject, and a call to `TipcMonClientTimeSetWatch(client_name, TRUE)` sets up watching for time information about an RTclient.

In a similar fashion, watching is turned off by calling the API function with the *watch_status* parameter set to FALSE (such as `TipcMonSubjectSubscribeSetWatch(subject_name, FALSE)`).

Calling a `TipcMonTypeSetWatch` function with the *watch_status* parameter set to TRUE causes a message of type `MON_TYPE_SET_WATCH` to be sent to RTserver, where *TYPE* is the kind of information being watched. This *TYPE* is very similar to the *Type* used in the name of the API function as shown in these examples:

- `TipcMonSubjectSubscribeSetWatch` sends a message of type `MON_SUBJECT_SUBSCRIBE_SET_WATCH`
- `TipcMonClientTimeSetWatch` sends a message of type `MON_CLIENT_TIME_SET_WATCH`

An RTserver may hold the information being watched or it may need to forward the message on to other RTservers or RTclients to watch the information. An example of information that an RTserver keeps track of is a list of subjects to which its direct RTclients are subscribing. An example of the information an RTserver does not know about is the information about an RTclient's message buffers.

If RTserver does not hold the information and RTclient has the information, RTclient must be reading and processing messages when the initial call to `TipcMonTypeSetWatch` is made. Messages can be read and processed through calls to `TipcSrvMainLoop`, `TipcSrvMsgNext`, `TipcMsgSearch`, or `TipcMsgSearchType` when the watching is enabled in order for the watching to take effect. If RTclient does not receive and process a `MON_TYPE_STATUS_SET_WATCH` message, it will not send out the watched information when it changes.

Whenever the information of interest changes, RTserver sends a message with the information to the program that set up the watch. The watched information is returned by RTserver using a `MON_TYPE_STATUS` message, where *TYPE* matches the *TYPE* in the `MON_TYPE_SET_WATCH` that initiated the watch. The message types are in pairs of `MON_TYPE_SET_WATCH` and `MON_TYPE_STATUS` as shown in these examples:

- `TipcMonSubjectSubscribeSetWatch` sends a message of type `MON_SUBJECT_SUBSCRIBE_SET_WATCH`, and a message of type `MON_SUBJECT_SUBSCRIBE_STATUS` is returned by RTserver whenever an RTclient starts subscribing to or stops subscribing to the subject of interest.
- `TipcMonClientTimeSetWatch` sends a message of type `MON_CLIENT_TIME_SET_WATCH`, and a message of type `MON_CLIENT_TIME_STATUS` is returned by RTserver whenever the time changes in the RTclient of interest.

A complete listing and description of these message types are described in [Watching Message Types](#) on page 413. The message grammar for these types are described in [Monitoring Message Types](#) on page 367.

Processing Watch Results

When the results of a watch are returned in a `MON_TYPE_STATUS` message, the message can be processed using an `RTclient` message process callback (created with `TipcSrvProcessCbCreate`). Using callbacks to process the status messages that are returned as the result of a watch being enabled is the recommended method for processing watch results. The main reason for this is that the messages arrive asynchronously. It is often unclear as to when or how often the information may change. The following section shows an example of how to process watch results using callbacks.

Using Callbacks

This example shows how to use an `RTclient` message process callback (created with `TipcSrvProcessCbCreate`) to process the message containing the results of a watch being set up:

```
/* ===== */
/*..process_mon_project_names_status - callback to process a
   MON_PROJECT_NAMES_STATUS message */
void T_ENTRY process_mon_project_names_status(
    T_IPC_CONN conn,
    T_IPC_CONN_PROCESS_CB_DATA data,
    T_CB_ARG arg)
{
    T_STR *project_names;
    T_INT4 num_project_names;
    T_STR created_project;
    T_STR destroyed_project;
    T_INT4 counter;

    /* access fields of status message */
    if (!TipcMsgSetCurrent(data->msg, 0)) {
        /* error */
    }

    if (!TipcMsgNextStrArray(data->msg, &project_names,
        &num_project_names)
        || !TipcMsgNextStr(data->msg, &created_project)
        || !TipcMsgNextStr(data->msg, &destroyed_project)) {
        /* error */
    }

    /* ...do whatever with results here */
} /* process_mon_project_names_status */
```

```

/* ===== */
/*...code from calling function is below */

T_IPC_MT mt;

/* setup watching of projects */
if (!TipcMonProjectNamesSetWatch(TRUE)) {
    /* error */
}

/* create callback to process MON_PROJECT_NAMES_STATUS messages */
mt = TipcMtLookupByNum(T_MT_MON_PROJECT_NAMES_STATUS);
if (mt == NULL) {
    /* error */
}

if (TipcSrvProcessCbCreate(mt, process_mon_project_names_status,
NULL)
    == NULL) {
    /* error */
}

/* At this point TipcSrvMainLoop can be used to read and process messages. */

```

Printing Watch Categories

The API function `TipcMonPrintWatch` can be used to print all the monitoring categories being watched by a program to a `TutOut`-style output function. The output function is called once for each line of output as shown in the example:

```

/* Set up some information to be watched */
if (!TipcMonSubjectSubscribeSetWatch(T_IPC_MON_ALL, TRUE) {
    /* error */
}
if (!TipcMonProjectNamesSetWatch(TRUE) {
    /* error */
}
if (!TipcMonClientTimeSetWatch("primary_rtie", TRUE) {
    /* error */
}
if (!TipcMonClientMsgRecvSetWatch("primary_rtie", "numeric_data",
TRUE) {
    /* error */
}
if (!TipcMonPrintWatch(TutOut)) {
    /* error */
}

```

The example prints this output:

```
Watching project_names.
Watching subject_subscribe <*>.
Watching client_time <primary_rtie>.
Watching client_msg_recv <primary_rtie> <numeric_data>.
```

Watching Message Types

This section contains a complete listing of each watching message type. For each message pair (*MON_TYPE_SET_WATCH* and *MON_TYPE_STATUS*), a description of the type of information, the API function, and where the information is collected from is shown.

The *T_MT_* prefix is omitted from the name of each message type for the sake of brevity.

MON_CLIENT_BUFFER_*

The *MON_CLIENT_BUFFER_SET_WATCH* message type is used to set whether or not an RTclient is watching message-related buffer information in one or more RTclients in a project. When watching an RTclient's message buffers, RTserver sends a *MON_CLIENT_BUFFER_STATUS* message each time the message queue for the connection to RTserver changes in the watched RTclients. When watching is first enabled, an initial status message is sent if the RTclient(s) being watched exists.

API Function: *TipcMonClientBufferSetWatch*

Message Initiated: *MON_CLIENT_BUFFER_SET_WATCH*

Message Returned: *MON_CLIENT_BUFFER_STATUS*

Info Gathered From: RTclient

MON_CLIENT_CONGESTION_*

The MON_CLIENT_CONGESTION_SET_WATCH message type is used to set whether or not an RTclient is watching for congestion in another RTclient's write buffer. When watching an RTclient's write buffers, RTserver sends a MON_CLIENT_CONGESTION_STATUS message if the number of pending messages in the write buffer reaches a set amount. Another message is sent when the number of pending messages decreases to a lower set amount.

API Function: TipcMonClientCongestionSetWatch

Message Initiated: MON_CLIENT_CONGESTION_SET_WATCH

Message Returned: MON_CLIENT_CONGESTION_STATUS

Info Gathered From: RTclient

MON_CLIENT_MSG_RECV_*

The MON_CLIENT_MSG_RECV_SET_WATCH message type is used to set whether or not an RTclient is watching message-received information in one or more RTclients in a project. When watching an RTclient's incoming messages, RTserver sends a MON_CLIENT_MSG_RECV_STATUS message each time a received message is inserted into or deleted from the message queue (for the connection to RTserver) in the watched RTclients. When watching is first enabled, an initial status message is sent if the RTclient(s) being watched exists.

API Function: TipcMonClientMsgRecvSetWatch

Message Initiated: MON_CLIENT_MSG_RECV_SET_WATCH

Message Returned: MON_CLIENT_MSG_RECV_STATUS

Info Gathered From: RTclient

MON_CLIENT_MSG_SEND_*

The MON_CLIENT_MSG_SEND_SET_WATCH message type is used to set whether or not an RTclient is watching sent messages in one or more RTclients in a project. When watching an RTclient's outgoing messages, RTserver sends a MON_CLIENT_MSG_SEND_STATUS message each time a message is sent to RTserver from the watched RTclients. When watching is first enabled, an initial status message is sent if the RTclient(s) being watched exists.

API Function: TipcMonClientMsgSendSetWatch

Message Initiated: MON_CLIENT_MSG_SEND_SET_WATCH

Message Returned: MON_CLIENT_MSG_SEND_STATUS

Info Gathered From: RTclient

MON_CLIENT_NAMES_*

The MON_CLIENT_NAMES_SET_WATCH message type is used to set whether or not an RTclient is watching RTclient names in a project. When watching RTclient names, RTserver sends a MON_CLIENT_NAMES_STATUS message each time an RTclient is created or destroyed. When watching is first enabled, an initial status message is sent if an RTclient is running. An RTclient is considered created when it connects to RTserver. An RTclient is considered destroyed when its connection to RTserver is closed or lost.

API Function: TipcMonClientNamesSetWatch

Message Initiated: MON_CLIENT_NAMES_SET_WATCH

Message Returned: MON_CLIENT_NAMES_STATUS

Info Gathered From: RTserver

MON_CLIENT_SUBSCRIBE_*

The MON_CLIENT_SUBSCRIBE_SET_WATCH message type is used to set whether or not an RTclient is watching the subjects to which one or more RTclients in a project are subscribing. When watching RTclient subscriptions, RTserver sends a MON_CLIENT_SUBSCRIBE_STATUS message each time the watched RTclients start or stop subscribing to a subject. When watching is first enabled, an initial status message is sent if the RTclient(s) being watched exists.

API Function: TipcMonClientSubscribeSetWatch

Message Initiated: MON_CLIENT_SUBSCRIBE_SET_WATCH

Message Returned: MON_CLIENT_SUBSCRIBE_STATUS

Info Gathered From: RTserver

MON_CLIENT_TIME_*

The MON_CLIENT_TIME_SET_WATCH message type is used to set whether or not an RTclient is watching time information in one or more RTclients in a project. When watching an RTclient's time information, RTserver sends a MON_CLIENT_TIME_STATUS message each time the current time changes in the watched RTclients. When watching is first enabled, an initial status message is sent if the RTclient(s) being watched exists.

API Function: TipcMonClientTimeSetWatch

Message Initiated: MON_CLIENT_TIME_SET_WATCH

Message Returned: MON_CLIENT_TIME_STATUS

Info Gathered From: RTclient

MON_PROJECT_NAMES_*

The MON_PROJECT_NAMES_SET_WATCH message type is used to set whether or not an RTclient is watching project names. When watching project names, RTserver sends a MON_PROJECT_NAMES_STATUS message each time a project is created or destroyed, as well as an initial status message when watching is first enabled.

API Function: TipcMonProjectNamesSetWatch

Message Initiated: MON_PROJECT_NAMES_SET_WATCH

Message Returned: MON_PROJECT_NAMES_STATUS

Info Gathered From: RTserver

MON_SERVER_CONGESTION_*

The MON_SERVER_CONGESTION_SET_WATCH message type is used to set whether or not an RTclient is watching for congestion in a process's write buffer. When watching the process's write buffers, RTserver sends a MON_SERVER_CONGESTION_STATUS message if the number of pending bytes in the write buffer reaches a set amount. Another message is sent when the number of pending bytes decreases to a lower set amount.

API Function: TipcMonServerCongestionSetWatch

Message Initiated: MON_SERVER_CONGESTION_SET_WATCH

Message Returned: MON_SERVER_CONGESTION_STATUS

Info Gathered From: RTserver

MON_SERVER_CONN_*

The MON_SERVER_CONN_SET_WATCH message type is used to set whether or not an RTclient is watching RTserver connections. When watching RTserver connections, RTserver sends a MON_SERVER_CONN_STATUS message each time a connection between RTservers is created or destroyed.

API Function: TipcMonServerConnSetWatch

Message Initiated: MON_SERVER_CONN_SET_WATCH

Message Returned: MON_SERVER_CONN_STATUS

Info Gathered From: RTserver

MON_SERVER_MAX_CLIENT_LICENSES_*

The MON_SERVER_MAX_CLIENT_LICENSES_SET_WATCH message type is used to set whether or not an RTserver is watching for the number of client connections to reach and exceeded the licensed amount. When watching client connection licenses, RTserver sends a MON_SERVER_MAX_CLIENT_LICENSES_STATUS message each time an RTserver refuses a client connection because no licenses were available.

API Function: TipcMonServerMaxClientLicensesSetWatch

Message Initiated: MON_SERVER_MAX_CLIENT_LICENSES_SET_WATCH

Message Returned: MON_SERVER_MAX_CLIENT_LICENSES_STATUS

Info Gathered From: RTserver

MON_SERVER_NAMES_*

The MON_SERVER_NAMES_SET_WATCH message type is used to set whether or not an RTclient is watching RTserver names. When watching RTserver names, RTserver sends a MON_SERVER_NAMES_STATUS message each time an RTserver is created or destroyed. An RTserver is considered created when it starts up (and usually connects to other RTservers). An RTserver is considered destroyed when it terminates or disconnects from other RTservers.

API Function: TipcMonServerNamesSetWatch

Message Initiated: MON_SERVER_NAMES_SET_WATCH

Message Returned: MON_SERVER_NAMES_STATUS

Info Gathered From: RTserver

MON_SUBJECT_NAMES_*

The MON_SUBJECT_NAMES_SET_WATCH message type is used to set whether or not an RTclient is watching subject names in a project. When watching subject names, RTserver sends a MON_SUBJECT_NAMES_STATUS message each time a subject is created or destroyed in the project. A subject is considered created when the first RTclient starts subscribing to that subject. A subject is considered destroyed when the last RTclient stops subscribing to that subject.

API Function: TipcMonSubjectNamesSetWatch

Message Initiated: MON_SUBJECT_NAMES_SET_WATCH

Message Returned: MON_SUBJECT_NAMES_STATUS

Info Gathered From: RTserver

MON_SUBJECT_SUBSCRIBE_*

The MON_SUBJECT_SUBSCRIBE_SET_WATCH message type is used to set whether or not an RTclient is watching the RTclients that are subscribing to a specified subject or any subject. When watching subjects being subscribed to, RTserver sends a MON_SUBJECT_SUBSCRIBE_STATUS message each time an RTclient starts or stops subscribing to the subject(s). When watching is first enabled, an initial status message is sent if the subject being watched exists.

API Function: TipcMonSubjectSubscribeSetWatch

Message Initiated: MON_SUBJECT_SUBSCRIBE_SET_WATCH

Message Returned: MON_SUBJECT_SUBSCRIBE_STATUS

Info Gathered From: RTserver

Watching Example

The following is a complete C source example that watches a subject and outputs information whenever an RTclient starts subscribing to a subject or stops subscribing to a subject.

The source code files for this example are located in these directories:

UNIX:

`$RTHOME/examples/smrtsock/manual`

OpenVMS:

`RTHOME : [EXAMPLES . SMRTSOCK . MANUAL]`

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

The online source files have additional #ifdefs to provide C++ support; these #ifdefs are not shown to simplify the example.

```
/*
-----
watchsbj.c

USAGE
watchsbj <project> <subject>

This program connects to RTserver and outputs a count and names
of RTclients that subscribed to <subject> in <project>.
Whenever an RTclient subscribes or unsubscribes to a subject,
information about the subject is output.
-----
*/

#include <rtworks/ipc.h>

/* ===== */
/*..cb_default - callback to process anything but
MON_SUBJECT_SUBSCRIBE_STATUS */
static void T_ENTRY cb_default(
    T_IPC_CONN conn,
    T_IPC_CONN_PROCESS_CB_DATA data,
    T_CB_ARG arg)
{
    TipcMsgPrintError(data->msg);
} /* cb_default */

/* ===== */
/*..process_subject_status - callback to process MON_SUBJECT_SUBSCRIBE_STATUS msg */
static void T_ENTRY process_subject_status(
    T_IPC_CONN conn,
    T_IPC_CONN_PROCESS_CB_DATA data,
    T_CB_ARG arg)
{
    T_IPC_MSG msg = data->msg;
    T_STR subject_name;
    T_STR *subscribe_client_names;
    T_INT4 num_subscribe_clients;
    T_STR start_client;
    T_STR stop_client;
    T_INT4 n;

    /* Set current field */
    if (!TipcMsgSetCurrent(msg, 0)) {
        TutOut("Could not set current field of message: error <%s>.\n",
            TutErrStrGet());
        return;
    }
}
```

```

/* Get the fields from the message */
if (!TipcMsgNextStr(msg, &subject_name)
    || !TipcMsgNextStrArray(msg, &subscribe_client_names,
                           &num_subscribe_clients)
    || !TipcMsgNextStr(msg, &start_client)
    || !TipcMsgNextStr(msg, &stop_client)) {
    TutOut("Unable to access MON_SUBJECT_SUBSCRIBE_STATUS");
    TutOut(" message: error <%s>.\n",
           TutErrStrGet());
    return;
}

/* Print out the new information extracted from message */
TutOut("Received change notice for subject <%s>\n",
subject_name);
TutOut("Number of clients subscribed to <%s> = %d\n",
       subject_name, num_subscribe_clients);
for (n = 0; n < num_subscribe_clients; n++) {
    TutOut("    [%d] %s\n", n, subscribe_client_names[n]);
}

TutOut("RTclient who just subscribed:  %s\n", start_client);
TutOut("RTclient who just unsubscribed: %s\n", stop_client);

TutOut("=====\n");
;

} /* process_subject_status */

/* ===== */
int main(argc, argv)
int argc;
char **argv;
{
    T_STR project_name;
    T_STR subject_name;
    T_IPC_MT mt;
    T_OPTION option;

/* Check the command line arguments */
if (argc != 3) {
    TutOut("Usage: watchsbj <project> <subject>\n");
    TutExit(T_EXIT_FAILURE);
}

/* Save pointers to command line arguments */
project_name = argv[1];
subject_name = argv[2];

TutOut("Watching subject <%s> in project <%s>...\n",
       subject_name, project_name);

```

```

/* Set the project name */
option = TutOptionLookup("project");
if (option == NULL) {
    TutOut("Could not look up option <project>: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

if (!TutOptionSetEnum(option, project_name)) {
    TutOut("Could not set option <project>: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* Connect to RTserver */
if (!TipcSrvCreate(T_IPC_SRV_CONN_FULL)) {
    TutOut("Could not connect to RTserver: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* Create callback to process subject status messages */
mt = TipcMtLookupByNum(T_MT_MON_SUBJECT_SUBSCRIBE_STATUS);
if (mt == NULL) {
    TutOut("Could not look up message type");
    TutOut(" MON_SUBJECT_SUBSCRIBE_STATUS: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

if (TipcSrvProcessCbCreate(mt, process_subject_status, NULL)
    == NULL) {
    TutOut("Could not create callback to process ");
    TutOut("MON_SUBJECT_SUBSCRIBE_STATUS: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* Create default callback to receive unwanted message types */
if (TipcSrvDefaultCbCreate(cb_default, NULL) == NULL) {
    TutOut("Could not create default callback: error <%s>.\n",
        TutErrStrGet());
}

/* Start "watching" the subject */
if (!TipcMonSubjectSubscribeSetWatch(subject_name, TRUE)) {
    TutOut("Could not start watching ");
    TutOut("subject <%s>: error <%s>.\n",
        subject_name, TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

```

```

/* If an error occurs, then TipcSrvMainLoop will restart RTserver */
/* and return FALSE. We can safely continue. */
for (;;) {
    if (!TipcSrvMainLoop(T_TIMEOUT_FOREVER)) {
        TutOut("TipcSrvMainLoop failed: error <%s>.\n",
            TutErrStrGet());
    }
}

/* This line should not be reached. */
return T_EXIT_FAILURE;
} /* main */

```

The call to `TipcMonSubjectSubscribeSetWatch` turns on watching for all RTclients subscribing to the specified subject. Whenever an RTclient starts or stops subscribing to the specified subject, RTserver asynchronously sends a `MON_SUBJECT_SUBSCRIBE_STATUS` message to the `watchsbj` program. Prior to turning watching on, a callback (`process_subject_status`) is created to process a message of this type when it arrives. The callback function `process_subject_status` then prints out the information contained in the message.

Initially, if there are no RTclients subscribing to the specified subject, no message is sent by RTserver to the program.

Compiling, Linking, and Running

To compile, link, and run the `watchsbj` program, first you must either copy the program to your own directory or have write permission in these directories:

UNIX:

```
$RTHOME/examples/smrtsock/manual
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]
```

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

You are also going to run a program to start subscribing to a subject.

Step 1 **Copy the `receive.c` program from one of these directories**

Note that this program is part of the `ss_tutorial` project and subscribes to the `lesson5` subject.

UNIX:

```
$RTHOME/examples/smrtsock/tutorial/lesson5
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.TUTORIAL.LESSON5]
```

Windows:

```
%RTHOME%\examples\smrtsock\tutorial\lesson5
```

Step 2 **Compile and link the `watchsbj` and `receive` programs**

UNIX:

```
$ rtlink watchsbj.c -o watchsbj.x
$ rtlink receive.c -o receive.x
```

OpenVMS:

```
$ cc watchsbj.c
$ rtlink /exec=watchsbj.exe watchsbj.obj
$ cc receive.c
$ rtlink /exec=receive.exe receive.obj
```

Windows:

```
$ nmake /f wsjw32m.mak
$ nmake /f recw32m.mak
```

To run the programs, start the `receive` process in one terminal emulator window and then the `watchsbj` process in another terminal emulator window.

Step 3 Start the receive program in the first window

UNIX:

```
$ receive.x
```

OpenVMS:

```
$ run receive.exe
```

Windows:

```
$ receive.exe
```

Step 4 Start the watchsbj program in the second window

UNIX:

```
$ watchsbj.x ss_tutorial lesson5
```

OpenVMS:

```
$ watchsbj := $sys$disk:[]watchsbj.exe
$ watchsbj ss_tutorial lesson5
```

Windows:

```
$ watchsbj.exe ss_tutorial lesson5
```

This is an example of the output displayed:

```
Watching subject <lesson5> in project <ss_tutorial>...
Connecting to project <ss_tutorial> on <_node> RTserver.
Using local protocol.
Message from RTserver: Connection established.
Start subscribing to subject </_grissom_1026>.
Received change notice for subject <lesson5>
/* This is the initial status message which is returned */
Number of clients subscribed to <lesson5> = 1
[0] /_grissom_1018
RTclient who just subscribed:
RTclient who just unsubscribed:
=====
```

Step 5 Kill the receive program and restart it

This is an example of the output displayed:

```
Received change notice for subject <lesson5>
Number of clients subscribed to <lesson5> = 0
RTclient who just subscribed:
RTclient who just unsubscribed: /_grissom_1018
=====
Received change notice for subject <lesson5>
Number of clients subscribed to <lesson5> = 1
[0] /_grissom_1028
RTclient who just subscribed: /_grissom_1028
RTclient who just unsubscribed:
=====
```

Note that `watchsbj` output results immediately when the program was killed and again when it was restarted.

Advanced Monitoring

This section describes advanced monitoring features, such as how to derive information, naming services, and using monitoring to implement a fault tolerant architecture.

Monitoring With SNMP

With your standard SmartSockets product, all you have to do is hook into a project, and all monitoring information for all processes is available. However, if you want to use SNMP products to monitor SmartSockets processes, you can purchase our SNMP support module, SmartSockets Monitor. For more information, see your TIBCO sales representative or the *TIBCO SmartSockets Monitor User's Guide*.

Deriving Information

In many situations, you may wish to derive (calculate) new information from the information received. This allows the process accessing the monitoring information to calculate any derived information it finds useful. One example of this is calculating the change in memory usage of an RTclient since the last poll. For example, the RTmon GDI calculates the change in memory usage from the last poll.

Process Identification

Each RTserver and RTclient has an identification string that is used as a descriptive name for the process when it is being monitored. This string shows up in RTmon and is also used as part of a field in these message types:

- MON_CLIENT_NAMES_STATUS
- MON_SERVER_NAMES_STATUS
- MON_CLIENT_NAMES_POLL_RESULT
- MON_SERVER_NAMES_POLL_RESULT

In the above message types, the full field has the form "ident: user@node" (such as "RTclient: ssuser@workstation1"). The monitoring identification string is used as ident. This string is retrieved with the function TipcMonGetIdentStr and set with the function TipcMonSetIdentStr, or the Monitor_Ident option. This example shows how to set and access the identification string of a process:

```
T_STR ident_str;

if (!TipcMonSetIdentStr("Acme Inc. Data Collector")) {
    /* error */
}

if (!TipcMonGetIdentStr(&ident_str)) {
    /* error */
}

TutOut("Monitoring identification string is %s\n", ident_str);
```

An RTclient that calls TipcMonSetIdentStr after calling TipcSrvCreate will not be identified correctly.

Naming (Directory) Services

SmartSockets monitoring provides an elegant way to implement naming and directory services within a project. Using monitoring combined with the publish-subscribe features of SmartSockets, it is very easy to find items of interest on a network. In general, RTclients are identified by the setting of their Unique_Subject option, and groups of RTclients can be identified by subscribing to a specified subject.

The monitoring functions that provide naming or directory services includes:

- `TipcMonClientGeneralPoll` — returns the node an RTclient is running on
- `TipcMonServerGeneralPoll` — returns the node an RTserver is running on

These functions, combined with other functions in the monitoring API, can be used to locate RTclients based on:

- the setting of their Unique_Subject option
- the subjects they are subscribing to
- the project they belong to
- the message types they know about
- the options they know about

For example, to locate what node an RTclient, identified by the unique subject `program_x`, resides on, this example is used:

```
T_IPC_MT mt;
T_STR client_name;
T_STR ident;
T_STR node_name;
T_STR user_name;
T_INT4 pid;
T_STR project;

/* send the poll request out to RTserver */
if (!TipcMonClientGeneralPoll("program_x")) {
    /* error */
}

mt = TipcMtLookupByNum(T_MT_MON_CLIENT_GENERAL_POLL_RESULT);
if (mt == NULL) {
    /* error */
}

/* wait up to 10 seconds for the poll result */
msg = TipcSrvMsgSearchType(10.0, mt);
if (msg == NULL) {
    /* error */
}
```

```

/* access the fields of interest from the returned message*/
if (!TipcMsgSetCurrent(msg, 0)) {
    /* error */
}

/* note that we do not have to access all the fields of the message */
if (!TipcMsgRead(data->msg,
                 T_IPC_FT_STR, &client_name,
                 T_IPC_FT_STR, &ident,
                 T_IPC_FT_STR, &node_name,
                 T_IPC_FT_STR, &user_name,
                 T_IPC_FT_INT4, &pid,
                 T_IPC_FT_STR, &project,
                 NULL)) {
    /* error */
}

/* Output the information */
TutOut("RTclient name = %s\n", client_name);
TutOut("ident = %s\n", ident);
TutOut("node name = %s\n", node_name);
TutOut("user name = %s\n", user_name);
TutOut("pid = %d\n", pid);
TutOut("project = %s\n", project);

```

Running an RTclient With a Hot Backup

In many mission-critical applications, fault tolerance and reliability are important requirements. The system requires continuous operation, 24 hours a day, 7 days a week, regardless of hardware or software failures. Handling Network Failures In Publish Subscribe on page 307, describes how RTserver and RTclient achieve increased reliability in their communication by transparently checking for problems that occur when connecting, sending messages, or receiving messages.

You can achieve a higher level of reliability in your system through the use of software redundancy. Redundancy involves one or more backup processes for each primary process. For example, if you want to ensure that a specified RTclient continues to run regardless of problems that may occur in the network, you can run one or more backup RTclients as a mirror for each primary RTclient.

To minimize down time, you typically want to run the backup RTclient in a hot backup mode. This means the backup RTclient runs in parallel with the primary RTclient, with the backup typically running on a different computer for increased reliability. When the primary RTclient goes down or fails to respond for a given amount of time, control is transferred to the backup RTclient, and it then takes over as the primary, possibly spawning a new backup for itself.

Another easy way to implement a backup RTclient process is to use the SORTED load balancing mode described in Chapter 3, Publish-Subscribe.

When running a backup RTclient in parallel with the primary RTclient, these requirements should be met so as to maximize continuous operation:

- The backup RTclient receives all the same data as the primary RTclient (they both start subscribing to the same subjects at the same time, and they both stop subscribing to the same subjects at the same time).
- The backup RTclient and the primary RTclient are functionally the same software
- The backup RTclient is silent. This means it does not send out any messages to other RTclients in the project. Only when an RTclient is the primary process can it send out its results using messages.

SmartSockets provides a straightforward mechanism to meet these requirements in running a primary RTclient with a hot backup. For example, if you want to ensure that an RTclient runs continuously, two RTclients (both identical programs) could be started on different machines with both subscribing to identical subjects. Both RTclients receive the same messages and make equivalent calculations, keeping their internal states consistent. The backup RTclient can run in silent mode by setting the standard option `Server_Msg_Send` to `FALSE`, thus preventing any messages from going out. Correspondingly, the primary RTclient should have its `Server_Msg_Send` option set to `TRUE`, ensuring its results are sent out. When the primary RTclient fails, the backup RTclient can have its `Server_Msg_Send` option set to `TRUE`, now making it the primary RTclient. Because there is now only a single RTclient running, a backup needs to be restarted and have its state restored to that of the primary.

To demonstrate this simple strategy of running redundant RTclients, one a mirror of the other, a complete example is shown and discussed in the following sections. This example uses a user-defined RTclient to monitor and control the primary and backup processes. This strategy is appealing in the fact that it is non-intrusive, allowing you to achieve the software fault tolerance without having to make any changes in the program used by the primary and backup RTclients.

Suppose that you want to run a hot backup for one RTclient in the project `user_manual`. The subjects to subscribe to are set in a command file that is read at startup. In this example, the RTclient is only interested in processing messages sent to the `chapter5` subject.

This is an example of the startup command file for the primary RTclient. The highlighted text is the text you must add to monitor the process for failure, and so failover can occur, if necessary:

```
setopt subjects      chapter5, primary_client
setopt server_msg_send true
```

The startup file for the backup RTclient is identical except it subscribes to the `backup_client` subject instead of `primary_client` subject, and it would have its `Server_Msg_Send` option set to `FALSE` as shown:

```
setopt subjects      chapter5, backup_client
setopt server_msg_send false
```

Note that in both command files the programs subscribe to the `chapter5` subject. Sending a message to the `chapter5` subject ensures that both the primary and backup RTclient receive the same message. The `backup_client` subject is reserved for messages that need to be sent to the hot backup only.

In the following example program, a user-defined RTclient (`guardian.c`) is used to perform the task of monitoring a primary and backup program. This program is referred to as the Guardian.

The source code files for this example are located in these directories:

UNIX:

```
$RTHOME/examples/smrtsock/manual
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]
```

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

The online source files have additional `#ifdefs` to provide C++ support; these `#ifdefs` are not shown to simplify the example.

Guardian (Hot Backup Manager) Source Code

This is the complete C source code example for the Guardian program:

```
/* guardian.c -- RTclient example for managing a hot backup process */
/*
```

```
USAGE
guardian.x <project name>
```

Also can use *udrecv.c*, *udsend.c*, *primary.cm*, and *backup.cm* to test.

This program is an RTclient that provides a primary and backup RTclient for the project passed in as a command-line argument. To use this, you must have two duplicate RTclients with different startup command files.

In the first command file, which is for the primary RTclient, the following lines should be added (or modified if they already exist):

```
setopt server_msg_send TRUE
setopt subjects <existing_subjects>, primary_client
```

The second command file, which is for the "hot backup", the following lines should be added (or modified if they already exist):

```
setopt server_msg_send FALSE
setopt subjects <existing_subjects>, backup_client
```

You should also have shell scripts to start the primary and backup RTclients named "startpcl" and "startbcl" respectively.

The guardian program then watches the primary_client and backup_client and switches the backup over as needed when the primary fails or restarts the backup if it fails.

```
*/
```

```
#define NOT_STARTED -1
#include <rtworks/ipc.h>
```

```
static T_STR project_name;
static T_INT4 num_primary = NOT_STARTED; /* Primary RTclient count */
static T_INT4 num_backup = NOT_STARTED; /* Backup RTclient count */
```

```
/* ===== */
/*..cb_default -- default callback to handle unexpected messages */
```

```
static void T_ENTRY cb_default(
    T_IPC_CONN conn,
    T_IPC_CONN_DEFAULT_CB_DATA data,
    T_CB_ARG arg)
{
    T_IPC_MT mt;
    T_STR name;
```

```

/* print out the name of the type of the message */
if (!TipcMsgGetType(data->msg, &mt)) {
    TutOut("Could not get message type from message: error
<%s>.\n",
        TutErrStrGet());
    return;
}
if (!TipcMtGetName(mt, &name)) {
    TutOut("Could not get name from message type: error <%s>.\n",
        TutErrStrGet());
    return;
}
TutOut("Unexpected message type name is <%s>\n", name);
} /* cb_default */

/* ===== */
/*..cb_subject_status -- callback to process MON_SUBJECT_SUBSCRIBE_STATUS messages */
static void T_ENTRY cb_subject_status(
    T_IPC_CONN conn,
    T_IPC_CONN_PROCESS_CB_DATA data,
    T_CB_ARG arg)
{
    T_IPC_MSG msg = data->msg;
    T_IPC_MT mt;
    T_STR subject_name;
    T_STR lc_subject_name; /* lower-case version of subject_name */
    T_INT4 subject_count;
    T_REAL8 wall_time;
    T_STR time_str;
    T_STR *subscribe_client_names;

/*
* cb_subject_status is called when a MON_SUBJECT_SUBSCRIBE_STATUS
* subject status message is received. Each time an RTclient starts
* or stops subscribing to the primary_client or backup_client
* subjects this function is called.
*/

/* Get the current wall clock time and convert it to a string */
    wall_time = TutGetWallTime();
    time_str = TutTimeNumToStr(wall_time);

/* Parse the subject status message for the subject id and */
/* the number of RTclients subscribed to it. */

/* Set current field */
    if (!TipcMsgSetCurrent(msg, 0)) {
        TutOut("Could not set current field of message: error <%s>.\n",
            TutErrStrGet());
        return;
    }
}

```

```

/* Get the fields of interest from the message */
if (!TipcMsgNextStr(msg, &subject_name)
    || !TipcMsgNextStrArray(msg, &subscribe_client_names,
                           &subject_count)) {
    TutOut("Unable to access MON_SUBJECT_SUBSCRIBE_STATUS");
    TutOut("message: error <%s>.\n", TutErrStrGet());
    return;
}

TutOut("Time = %s\n", time_str );
TutOut("%d RTclients are subscribing to the %s subject.\n",
       subject_count, subject_name );

/* make a copy of the subject name and convert to lower case */
T_STRDUP(lc_subject_name, subject_name);
TutStrLwr(lc_subject_name, lc_subject_name);

if (strcmp(lc_subject_name, "primary_client") == 0) {
    num_primary = subject_count;
}
else if (strcmp(lc_subject_name, "backup_client") == 0) {
    num_backup = subject_count;
}
else {
    TutOut("Received SUBJECT_SUBSCRIBE_STATUS for unwanted");
    TutOut("subject %s.\n", lc_subject_name);
    T_FREE(lc_subject_name);
    return;
}

/* Prepare to send CONTROL messages later on. */
mt = TipcMtLookupByNum(T_MT_CONTROL);
if (mt == NULL) {
    TutOut("Could not look up control message: error <%s>.\n",
          TutErrStrGet());
    return;
}

```

```

/*
 * There are four main cases that need to be covered. They are
 * listed below followed by the actions to be taken when the case
 * is encountered:
 *   CASE 1: Neither primary nor backup RTclient is running.
 *     start primary RTclient
 *     start backup RTclient
 *   CASE 2: Both primary and backup RTclients are running.
 *     output message that all is OK
 *   CASE 3: Primary RTclient fails; backup RTclient is running.
 *     set server_msg_send option in backup RTclient to TRUE
 *     Have backup start subscribing to the primary_client subject
 *     Have backup stop subscribing to the backup_client subject
 *     (This will then cause CASE 4 to occur);
 *   CASE 4: Backup RTclient fails, primary RTclient is running.
 *     Restart backup RTclient
 */

/* Check if neither the primary RTclient nor backup RTclient have */
/* been started yet */
if (num_primary == 0 && num_backup == 0) {
    TutOut("Neither primary nor backup RTclient yet started.\n");
    TutOut("Starting primary RTclient...\n");
    TutSystem("startpcl &");
    TutOut("Starting backup RTclient...\n");
    TutSystem("startbcl &");
}

else if (num_primary == 1 && num_backup == 1) {
    TutOut("Both primary and backup RTclients are running!\n");
}

else if (strcmp(lc_subject_name, "primary_client") == 0) {
    if (num_primary == 1 && num_backup <= 0) {
        TutOut("Primary RTclient up and running! Waiting on ");
        TutOut("backup...\n");
    }
    else if (num_primary == 0 && num_backup <= 0) {
        TutOut("No primary RTclient yet; No report yet from backup\n");
    }
}

```

```

/* Check if we have lost the primary RTclient */
else if (num_primary == 0 && num_backup == 1 ){
    TutOut("Primary RTclient has failed!\n");

    TutOut("Switching the backup RTclient to be primary...\n");
    if (!TipcSrvMsgWrite("backup_client", mt, TRUE, T_IPC_FT_STR,
        "setopt server_msg_send TRUE", NULL)) {
        TutOut("Could not send setopt control message to ");
        TutOut("backup_client: error <%s>.\n", TutErrStrGet());
    }

    if (!TipcSrvMsgWrite("backup_client", mt, TRUE, T_IPC_FT_STR,
        "subscribe primary_client", NULL)) {
        TutOut("Could not send subscribe control message to ");
        TutOut("backup_client: error <%s>.\n", TutErrStrGet());
    }

    if (!TipcSrvMsgWrite("backup_client", mt, TRUE, T_IPC_FT_STR,
        "unsubscribe backup_client", NULL)) {
        TutOut("Could not send unsubscribe control message to ");
        TutOut("backup_client: error <%s>.\n", TutErrStrGet());
    }
}
else {
    TutOut("We have an irregular number of RTclients!\n");
    TutOut("Number of primary RTclients: %d\n", num_primary);
    TutOut("Number of backup RTclients: %d\n", num_backup);
}
}
else if (strcmp(lc_subject_name, "backup_client") == 0) {
    if (num_primary <= 0 && num_backup == 1) {
        TutOut("Backup RTclient up and running! Waiting on ");
        TutOut("primary...\n");
    }
    else if (num_primary <= 0 && num_backup == 0) {
        TutOut("No backup RTclient yet; ");
        TutOut("No report received yet from primary RTclient.\n");
    }
}
/* Check if we have lost the backup RTclient */
else if (num_primary == 1 && num_backup == 0){
    TutOut("Backup RTclient is down!\n");
    TutOut("Starting a new backup RTclient!\n");
    TutSystem("startbcl &");
}
else {
    TutOut("We have an irregular number of RTclients!\n");
    TutOut("Number of primary RTclients : %d\n",
        num_primary);
    TutOut("Number of backup RTclients : %d\n",
        num_backup);
}
}

TutOut("=====\n");
T_FREE(lc_subject_name);
} /* cb_subject_status */

```

```

/* ===== */
/*..main -- main program */
int main(argc, argv)
int argc;
char **argv;
{
    T_OPTION option;
    T_IPC_MT mt;

    /* Check the command-line arguments */
    if (argc != 2) {
        TutOut("Usage: guardian <project>\n");
        TutExit(T_EXIT_FAILURE);
    }

    /* Save the pointer to the command line argument */
    project_name = argv[1];
    TutOut("Monitoring project <%s>...\n", project_name);

    /* Set the project name */
    option = TutOptionLookup("project");
    if (option == NULL) {
        TutOut("Could not look up option named project: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
    if (!TutOptionSetEnum(option, project_name)) {
        TutOut("Could not set the option named <project>: error
<%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }

    /* Set the time format for the FULL format */
    TutCommandParseStr("setopt time_format full");

    /* Create a connection to RTserver */
    if (!TtipcSrvCreate(T_IPC_SRV_CONN_FULL)) {
        TutOut("Could not connect to RTserver: error <%s>.\n",
            TutErrStrGet());
        TutExit(T_EXIT_FAILURE);
    }
}

```

```

/* Create callback to process MON_SUBJECT_SUBSCRIBE_STATUS msgs */
mt = TipcMtLookupByNum(T_MT_MON_SUBJECT_SUBSCRIBE_STATUS);
if (mt == NULL) {
    TutOut("Could not look up MON_SUBJECT_SUBSCRIBE_STATUS");
    TutOut("message type: error <%s>.\n", TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (TipcSrvProcessCbCreate(mt, cb_subject_status, NULL) == NULL)
{
    TutOut("Could not create MON_SUBJECT_SUBSCRIBE_STATUS");
    TutOut("process callback: error <%s>.\n", TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* Create default callback to handle unwanted message types */
if (TipcSrvDefaultCbCreate(cb_default, NULL) == NULL) {
    TutOut("Could not create default callback: error <%s>.\n",
        TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* Start watching primary_client and backup_client subjects */
TutOut("Starting to watch <primary_client> subject.\n");
if (!TipcMonSubjectSubscribeSetWatch("primary_client", TRUE)) {
    TutOut("Could not start watching primary_client subject.\n");
    TutOut(" error <%s>.\n", TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

TutOut("Starting to watch <backup_client> subject.\n");
if (!TipcMonSubjectSubscribeSetWatch("backup_client", TRUE)) {
    TutOut("Could not start watching backup_client subject.\n");
    TutOut(" error <%s>.\n", TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

/* If RTserver stops, then TipcSrvMainLoop will restart RTserver */
/* and return FALSE. We can safely continue. */
for (;;) {
    if (!TipcSrvMainLoop(T_TIMEOUT_FOREVER)) {
        TutOut("TipcSrvMainLoop failed: error <%s>.\n",
            TutErrStrGet());
    }
}

/* This line should never be reached */
TutOut("This line should never be reached!\n");
return T_EXIT_FAILURE;
} /* main */

```


Compiling, Linking, and Running

To compile, link, and run the `guardian` program, first you must either copy the program to your own directory or have write permission in these directories:

UNIX:

```
$RTHOME/examples/smrtsock/manual
```

OpenVMS:

```
RTHOME:[EXAMPLES.SMRTSOCK.MANUAL]
```

Windows:

```
%RTHOME%\examples\smrtsock\manual
```

In addition to the `guardian` program, these two other programs are provided so you can test the application:

- `udrecv.c`

This program outputs field of `COUNT` message (an integer) and sends it back to `udsend.c` (two instances of this program running to simulate a primary and backup `RTclient`). This program takes a single command line argument, the name of a command file. In the test these command files are `primary.cm` and `backup.cm`.

- `udsend.c`

This program sends a `COUNT` message (an integer, incremented each time) to the `chapter5` subject; also has a callback to output the field of any `COUNT` messages sent back to it from `udrecv.c`. `COUNT` is a user-defined message type that consists of one field, an integer. The program `udsend` increments this one time, each time it sends a message.

Step 1 Compile and link all three programs, `guardian`, `udrecv`, and `udsend`

UNIX:

```
$ rtlink -o guardian.x guardian.c
$ rtlink -o udrecv.x udrecv.c
$ rtlink -o udsend.x udsend.c
```

OpenVMS:

```
$ cc guardian.c
$ rtlink /exec=guardian.exe guardian.obj
$ cc udrecv.c
$ rtlink /exec=udrecv.exe udrecv.obj
$ cc udsend.c
$ rtlink /exec=udsend.exe udsend.obj
```

Windows:

```
$ nmake /f gardw32m.mak
$ nmake /f udrcw32m.mak
$ nmake /f udsdw32m.mak
```

Step 2 Run the stguardn command

To run the test, run the `stguardn` command that starts guardian, two instances of `udrecv` (one primary and one backup), and one `udsend`.

UNIX:

```
$ stguardn user_manual
```

OpenVMS:

```
$ @stguardn user_manual
```

Windows:

```
$ stguardn user_manual
```

This is an example of the output:

```
Monitoring project <user_manual>...
Connecting to project <user_manual> on <_node> RTserver.
Using local protocol.
Message from RTserver: Connection established.
Start subscribing to subject </_workstation1_2252>.
Starting to watch <primary_client> subject.
Starting to watch <backup_client> subject.
Time = Fri Mar 14 12:54:50.567 1997
1 RTclients are subscribing to the primary_client subject.
Primary RTclient up and running! Waiting on backup...
=====
Time = Fri Mar 14 12:54:50.689 1997
1 RTclients are subscribing to the backup_client subject.
Both primary and backup RTclients are running!
=====
```

Step 3 Test failover by killing udrecv

Once both the primary and backup udrecv programs are running, along with udsend, test the failover by killing the primary udrecv or backup udrecv programs.

If the primary udrecv exits, then Guardian produces output similar to:

```
Time = Fri Mar 14 12:55:37.929 1997
0 RTclients are subscribing to the primary_client subject.
Primary RTclient has failed!
Switching the backup RTclient to be primary...
=====
Time = Fri Mar 14 12:55:38.138 1997
1 RTclients are subscribing to the primary_client subject.
Both primary and backup RTclients are running!
=====
Time = Fri Mar 14 12:55:38.180 1997
0 RTclients are subscribing to the backup_client subject.
Backup RTclient is down!
Starting a new backup RTclient!
=====
Time = Fri Mar 14 12:55:42.926 1997
1 RTclients are subscribing to the backup_client subject.
Both primary and backup RTclients are running!
=====
```

Once both the primary and backup udrecvs are running, if the backup udrecv exits, then Guardian produces output similar to:

```
Time = Fri Mar 14 12:55:22.772 2002
0 RTclients are subscribing to the backup_client subject.
Backup RTclient is down!
Starting a new backup RTclient!
=====
Time = Fri Mar 14 12:55:27.752 2002
1 RTclients are subscribing to the backup_client subject.
Both primary and backup RTclients are running!
=====
```

Discussion of the Guardian Program

Guardian uses the function `TipcMonSubjectSubscribeSetWatch` to set up subscription monitoring on the `primary_client` and `backup_client` subjects:

```
if (!TipcMonSubjectSubscribeSetWatch("primary_client", TRUE)) {
    TutOut("Could not start watching primary_client subject.\n");
    TutOut("  error <%s>.\n", TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}

if (!TipcMonSubjectSubscribeSetWatch("backup_client", TRUE)) {
    TutOut("Could not start watching backup_client subject.\n");
    TutOut("  error <%s>.\n", TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

Whenever the subscriber count for the `primary_client` subject changes, `RTserver` sends a `MON_SUBJECT_SUBSCRIBE_STATUS` message to Guardian. An `RTclient` message process callback function must be created for this message type within Guardian:

```
mt = TipcMtLookupByNum(T_MT_MON_SUBJECT_SUBSCRIBE_STATUS);
if (mt == NULL) {
    TutOut("Could not look up MON_SUBJECT_SUBSCRIBE_STATUS");
    TutOut("message type: error <%s>.\n", TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
if (TipcSrvProcessCbCreate(mt, cb_subject_status, NULL) == NULL) {
    TutOut("Could not create MON_SUBJECT_SUBSCRIBE_STATUS");
    TutOut("process callback: error <%s>.\n", TutErrStrGet());
    TutExit(T_EXIT_FAILURE);
}
```

Whenever an `RTclient` starts or stops subscribing to either the `primary_client` or `backup_client` subjects, a `MON_SUBJECT_SUBSCRIBE_STATUS` message is sent by `RTserver` to Guardian, causing the callback function `cb_subject_status` to be executed. The function `cb_subject_status` provides the failover and starting of the `RTclients`.

There are five cases that `cb_subject_status` must be able to handle:

1. Neither the primary nor the backup `RTclient` is running. In this case `cb_subject_status` should start both the primary and backup `RTclients`.

```
TutSystem("startpcl &");
TutSystem("startbcl &");
```

The shell scripts `startpcl` and `startbcl` start the respective processes.

2. Both the primary and backup `RTclients` are running. In this case, operation is normal and nothing needs to be done.

3. The primary RTclient fails, and the backup RTclient is running. This is the most interesting case as it requires the backup RTclient to be changed to the primary RTclient. To do this, three actions are required:

- a. A CONTROL message must be sent to the backup RTclient to set its `Server_Msg_Send` option to TRUE. This activates the backup, causing it to now start sending its messages to other RTclients.

```
if (!TipcSrvMsgWrite("backup_client", mt, TRUE, T_IPC_FT_STR,
                    "setopt server_msg_send TRUE", NULL)) {
    TutOut("Could not send setopt control message to ");
    TutOut("backup_client: error <%s>.\n", TutErrStrGet());
}
```

- b. The next step involves sending a CONTROL message to the backup RTclient to start subscribing to the `primary_client` subject. In essence, this makes the backup now the primary RTclient:

```
if (!TipcSrvMsgWrite("backup_client", mt, TRUE, T_IPC_FT_STR,
                    "subscribe primary_client", NULL)) {
    TutOut("Could not send subscribe control message to ");
    TutOut("backup_client: error <%s>.\n", TutErrStrGet());
}
```

- c. At this time, the RTclient is subscribing to messages for both the `primary_client` and `backup_client` subjects. To complete the hot failover, a CONTROL message is sent to the backup RTclient to stop subscribing to the `backup_client` subject. This completes the transition of the backup to the primary.

When the `backup_client` subject is no longer being subscribed to, a new `MON_SUBJECT_SUBSCRIBE_STATUS` message is issued by RTserver, specifying the process count for the `backup_client` subject is now zero, thus causing case 4 (described below) to occur.

```
if (!TipcSrvMsgWrite("backup_client", mt, TRUE, T_IPC_FT_STR,
                    "unsubscribe backup_client", NULL)) {
    TutOut("Could not send unsubscribe control message to ");
    TutOut("backup_client: error <%s>.\n", TutErrStrGet());
}
```

4. The primary RTclient is running and the backup fails. In this case, another backup RTclient must be started.

```
TutSystem("startbcl &")
```

The ampersand (&) is critical in the call to `TutSystem` as it starts the process in the background (in OpenVMS this is called spawning the process) and returns immediately, without waiting for it to complete. Failure to specify the ampersand (&) causes the Guardian process to hang.

5. Otherwise case: If any of the above cases do not handle the `MON_SUBJECT_SUBSCRIBE_STATUS` message, then an irregular number of RTclients is running, and an error message is displayed.

A key question is whether one can lose any outgoing messages from using this approach. The only time this happens is from the time RTserver detects the loss of the primary RTclient to the time that Guardian is able to switch the backup RTclient to be primary. This time period in most cases should be less than one second. If the primary RTclient fails at the same time it is sending out messages, there is potential loss of data. This risk can be further reduced by using guaranteed message delivery. See Chapter 4, *Guaranteed Message Delivery*, for more details.

Chapter 6 Using RTmon

This chapter presents detailed information about the tools available for monitoring and managing your SmartSockets project.

Topics

- *The RTmon Process, page 446*
- *RTmon Graphical Development Interface, page 447*
- *Monitoring Your Project with RTmon GDI, page 453*
- *Sending Messages with RTmon GDI, page 471*
- *Stopping RTmon GDI Processes, page 474*
- *RTmon Command Interface, page 475*

The RTmon Process

You can monitor and manage your entire SmartSockets project using an RTmon process. The RTmon is a standard RTclient that is non-intrusive, allowing you to monitor and manage information about your project without changing the running processes. RTmon also provides real-time system usage information on processes, such as CPU and memory resources.

You can access RTmon using the RTmon Graphical Development Interface (GDI) or through a built-in command-oriented interface called the RTmon Command Interface (CI).

The RTmon GDI is a graphical point-and-click tool that is intuitive and easy to use. RTmon supports many management sessions concurrently, providing real-time information using multiple RTclient windows. This chapter presents detailed information about invoking and using the RTmon GDI.



The RTmon GDI has been deprecated and may be removed in a future release.

The CI, also referred to as RTmon runtime, is a command-oriented interface for monitoring and managing your SmartSockets project.

The RTmon GDI runs as a separate process from the CI. Commands initiated in the RTmon GDI are sent to RTmon CI to be executed. The RTmon CI, in turn, calls on the TipcMon* API to execute the command. For example, initiating a poll command executes a TipcMonTypePoll function, and initiating a watch command executes a TipcMonTypeSetWatch function. Results are returned in standard monitoring message types to the RTmon CI and then sent on to the RTmon GDI for display.

It is assumed you are already familiar with using standard GDIs, including mouse functions, graphical views, hierarchical pull-down menus, and so on.

RTmon Graphical Development Interface



The RTmon GDI has been deprecated and may be removed in a future release.

The RTmon Graphical Development Interface (GDI) is a graphical point-and-click interface for monitoring and managing processes and message-related information in a distributed project. The RTmon GDI provides easy access to many of the commands discussed in Chapter 9, Command Reference. In addition, the GDI provides an assortment of tools for viewing projects, including graphical trees, browsers, and watch windows.

The RTmon GDI and the RTmon CI are actually two separate processes that communicate with each other using messages. In general, as you execute a GDI function, commands are formatted and sent to the CI. Any output produced by the execution of the command is then sent from the CI process back to the GDI process. For more information about starting RTmon CI, see RTmon Command Interface on page 475.

The content and function of the RTmon GDI on the Windows and Motif platforms are identical.

Starting a Graphical Development Interface Session

This section presents steps on how to invoke an RTmon GDI session, as well as basic information about using the RTmon GDI. Type this command at the prompt to invoke the RTmon GDI:

```
$ rtmon
```

Once the RTmon is finished initializing, the SmartSockets RTmon main window appears, as shown in Figure 35. If a project is not explicitly set in the `rtmon.cm` file, then an information window appears providing you with instructions. When you select OK in the Information box, the Project Browser pop-up box appears and allows you to select a project. Then the main window appears:

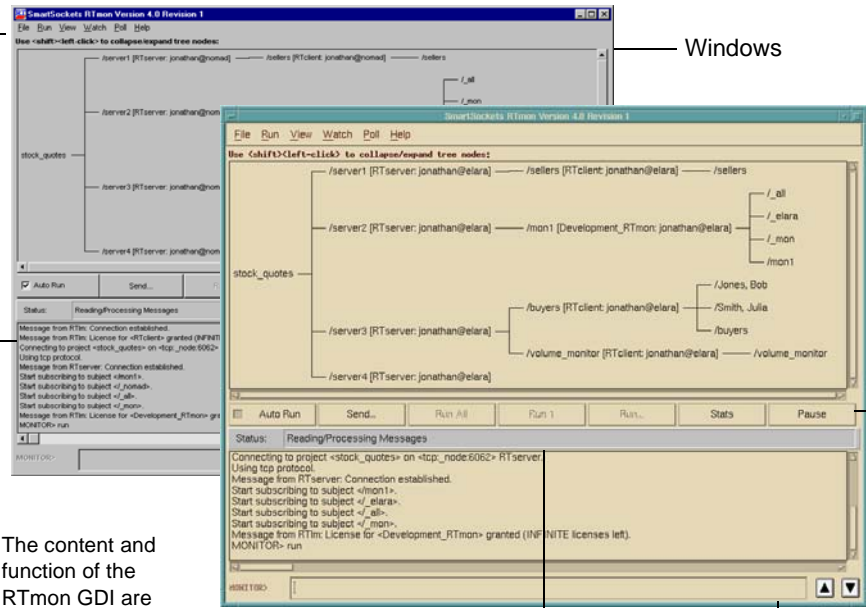
Figure 35 RTmon Main Window

The Menu Bar

Provides access to the most commonly used RTmon tools.

Output Command Interface

Displays project status information.



Windows

Motif

Command Bar

Provides point-and-click command buttons for the most commonly used

The content and function of the RTmon GDI are identical between the Windows and Motif platforms.

Status Line

Displays the current status of the runtime process.

Input Command Interface

Provides a command line for user input. This is only enabled when the GDI is paused.

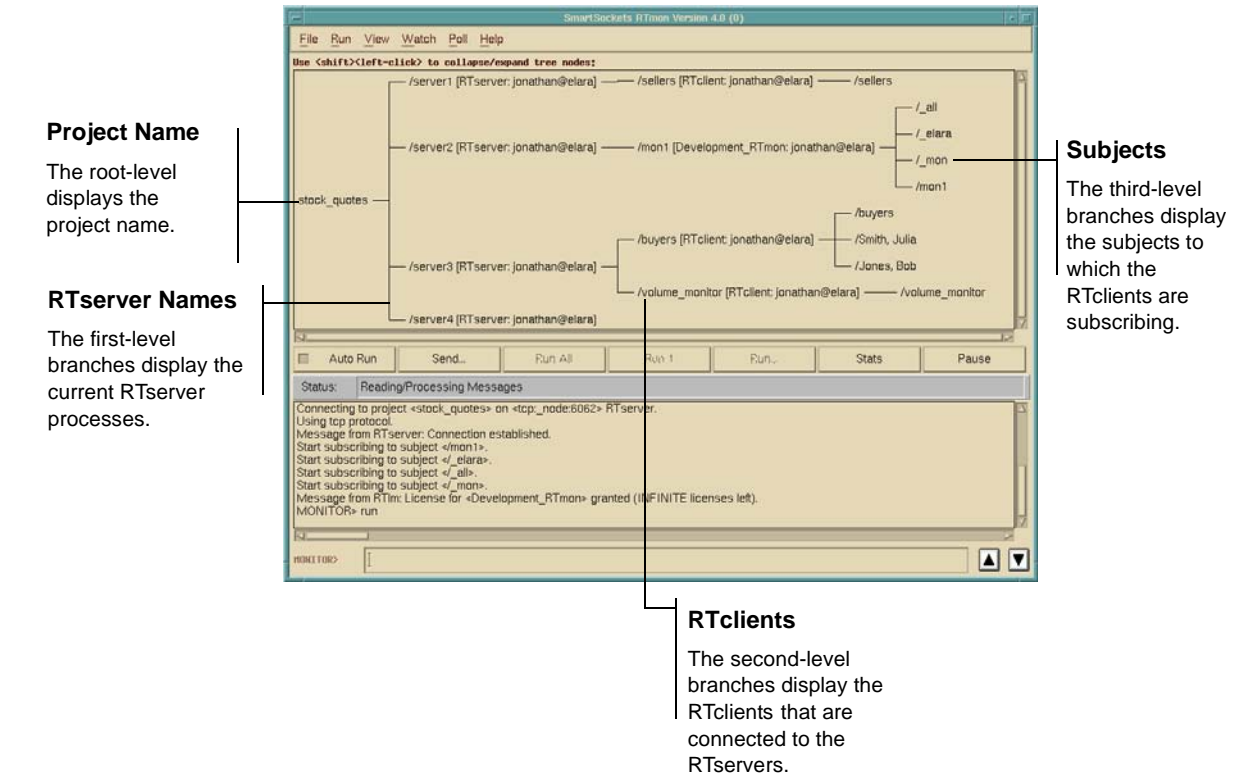
The six major functional regions of the RTmon Main window are:

Menu Bar	Provides point-and-click access to the most often used tools in an RTmon project.
Project Tree	Displays the processes that make up a project. The Project Tree is displayed in a directory tree format, and is updated in real time. The root of the tree represents the project name; the first-level branches from the project represent the RTserver processes, and the second-level branches represent the RTclient processes attached to the RTserver. The third-level branches (the outermost branches) represent the subjects to which each RTclient process is subscribing.
Command Bar	Displays the most commonly used commands in a command bar format.
Status Line	Displays the current status of the RTmon process.
Command Interface Output	Displays status information about the RTmon runtime process. Results from user-entered commands are displayed in this region.
Command Interface Input	Provides a command line for user input for the RTmon runtime process. Results are then displayed in the Command Interface Output region.

Vertical and horizontal scroll bars are available for scrolling the Project Tree, the Command Interface Output, and the Command Interface Input regions.

Figure 36 highlights the branches of the Project Tree.

Figure 36 RTmon Main Window Main Functional Regions



Menu Bar

- The menu bar provides access to six pull-down menus:
- File** Edits the `rtmon.cm` file and exits the RTmon GDI.
 - Run** Runs one or more messages, provides process statistics, and pauses any running processes.
 - View** Clears the command output interface, toggles auto scrolling, or modifies the view format in the Project Tree.
 - Watch** Monitors a project, RTclient time, RTclient message buffers, RTclients subscribing to subjects, subjects being subscribed to by RTclients, RTclient messages received, RTclient messages sent, and RTserver connections. This information is updated as changes occur.

- | | |
|------|---|
| Poll | Makes a one-time request for monitoring information about RTclients, RTservers, message traffic related to a specific RTclient or subject, or an RTserver buffer. |
| Help | Lists all available commands or options in the terminal window from which RTmon was started. |

The RTmon GDI is intuitive and easy to use as it is designed like many standard graphical interfaces. The following list presents some of the standard design features.

- Menu options followed by a right-angled triangle indicate one or more child pull-down menus.
- Items followed by an ellipsis indicate one or more child-windows.
- Items preceded by a square indicate a toggle option (i.e., selecting the menu option disables or enables the feature).
- In many windows where lists are presented, multiple items in the list may be selected simultaneously. To select more than one list item on Motif, highlight an item and then press the Shift key while left clicking on one or more of the other items in the list. To select more than one list item on Windows, highlight an item and then press the Ctrl key while left clicking on one or more of the other items in the list.

Project Tree

The Project Tree displays branches made up of the current project name, RTserver names, RTclient names, and subject names. The project name is the root of the tree. Branching off from the root are the RTserver processes, branching off of each RTserver are the RTclient processes connected to that RTserver, and branching off from each RTclient are the subjects to which that RTclient is subscribing. This window is updated in real time as changes occur to RTservers, RTclients, or subjects. For example, when a process subscribes or unsubscribes to a subject, the subject portion of the tree for that process is updated immediately. The Project Tree is useful for concise, up-to-date graphical snapshots of the project.

The Project Tree provides hot spots for accessing child windows. Double left click on any item displayed in the tree and a pop-up window appears. For instance, if you double left click on the project name, the Project Browser window appears. Similarly, if you double left click on one of the subjects displayed in the Project Tree, the Client Subjects Message Traffic window appears. Note that when you invoke a window from the Project Tree a poll is executed automatically.

Command Bar

The Command Bar is located just beneath the Project Tree region, providing point-and-click access to some of the most frequently used commands of the RTmon GDI.

Status Line

The Status Line is located immediately below the Command Bar region, presenting the current state of the RTmon process. The possible states include:

- Reading/Processing Messages
- Waiting For Command



The command interface saves the output of the session into a buffer indefinitely, consuming memory. To clear the memory buffer, select **View > Clear Output** from the Main menu.

Command Interface Output Region

The Command Interface Output Region displays standard output for the RTmon process. The output area automatically scrolls each time new output is received. This behavior may be bothersome if you are trying to look back at a message previously written to the screen.

Select **View>Auto Scroll** from the Main menu to change the default scrolling behavior. When auto scrolling is turned off, new output is written below the area that is currently displayed.

Command Interface Input Region

The Command Interface Input Region provides a command line for manually entering RTmon commands. Though the most common commands for running and watching a project can be accessed through the Main menu bar or through the Command Bar, most of the standard RTmon textual commands are available using this command interface. The following commands are not available through the GDI Command Interface Input region and must be executed from the RTmon CI.

- `poll` command
- `watch` command
- `unwatch` command

For more information about the textual commands see Chapter 9, Command Reference.

Monitoring Your Project with RTmon GDI



The RTmon GDI has been deprecated and may be removed in a future release.

The RTmon GDI has many built-in child windows for monitoring and managing specifics about your project.

This section is more helpful if you read it as you walk through a session with the RTmon GDI on your system.

Information Windows

Throughout the RTmon session, informational windows appear alerting you when an urgent condition occurs or providing instructions. The most commonly seen informational window appears when RTmon is invoked before a project is specified in the `rtmon.cm` file, as described in Starting a Graphical Development Interface Session on page 448.

After reading the information in provided in the window, click OK to close the window and move on.

Selecting a Project to Monitor

Select **Watch > Project Name** from the Main menu to choose a new project. The Project Browser window appears. The Project Browser window presents a list of the current projects known by the RTserver processes. Highlight the project you want to monitor. Select Change Project or double click on the project name to select that project. Click on Close to close the window.

Selecting a Command File

You can source a command file (**File > Source Cmd File**), edit an existing command file (**File > Edit Cmd File**), or save a command file (**File > Save Cmd File**). After making your menu selection, a Command File window appears. To select the command file, choose a directory and filename, then click OK.

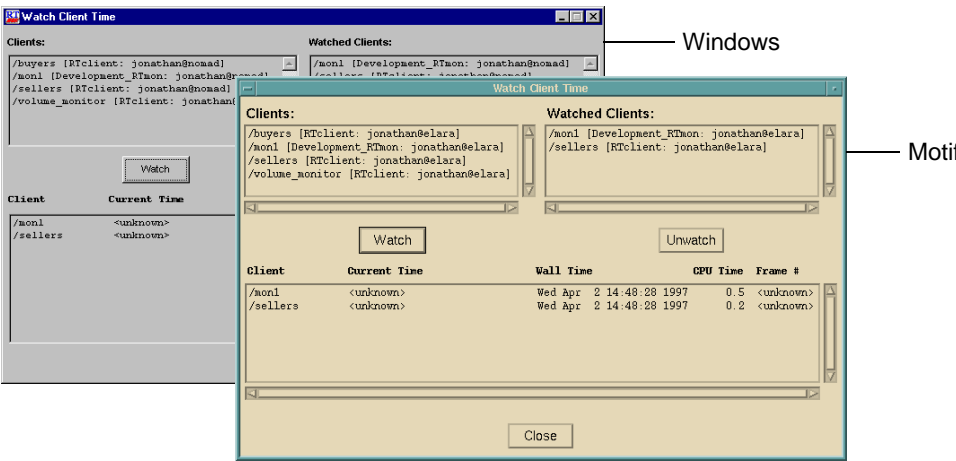
Monitoring RTclients

Checking Synchronization

Select **Watch > Client Time...** from the Main menu to view time information for one or more of the RTclients in your project. The Watch Client Time window appears. This is useful for observing whether all the RTclients are synchronized. The time information displayed in this window is updated in real time, so changes are displayed as they occur.

To watch an RTclient, highlight one or more of the RTclients displayed in the Clients list, then left click on the Watch button. To stop watching an RTclient, highlight the RTclient displayed in the Watched Clients list, and left click on the Unwatch button.

Figure 37 Watch Client Time Window



Checking Buffer and Queue Size

Select **Watch > Client Buffer...** from the Main menu to view buffer information for one or more of the RTclients in your project. The Watch Client Buffer window appears. This is useful for viewing the read/write buffer and message queue size (in bytes and number of messages). This window is similar to the Watch Client Time window. The buffer information is displayed in real time, and you can use the Watch and Unwatch buttons to select an RTclient for watching or to stop watching.

Checking Subjects Being Subscribed To

Select **Watch > Client Subjects...** from the Main menu to view the subject subscription information for one or more of the RTclients in your project. The Watch Clients Subjects window appears. This is useful for monitoring which subjects your RTclients are subscribing to. Again, the subject information is displayed in real time, and you can use the Watch and Unwatch buttons to select an RTclient for watching or to stop watching.

Monitoring Subjects

Checking Clients Subscribed to a Subject

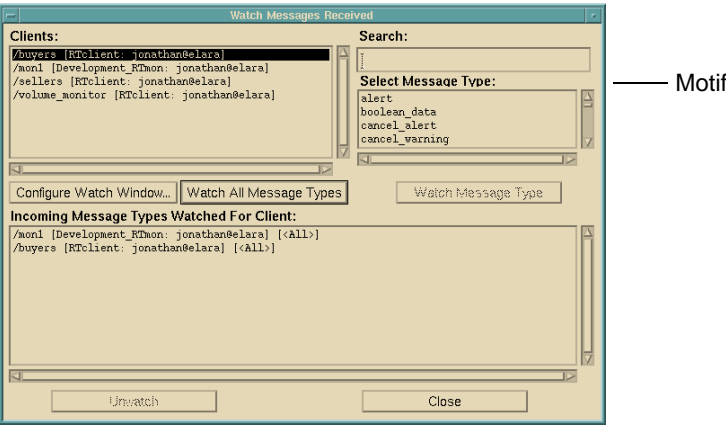
Select **Watch > Subjects Subscribed...** from the Main menu to view the RTclients subscribed to one or more subjects. The Watch Clients in Subject window appears. Instead of seeing the subjects that selected RTclients are subscribed to, you see the RTclients listed by the subjects they subscribe to. The RTclient information displayed in this window is updated in real time.

To watch a subject, highlight one or more of the subjects displayed in the Subjects list, then left click on the Watch button. To stop watching a subject, highlight the subject displayed in the Watched Subjects list, and left click on the Unwatch button.

Monitoring Messages Being Received

Select **Watch > Messages Received...** from the Main menu to display the Watch Messages Received window (shown in Figure 38).

Figure 38 Watched Messages Received Window



This window displays the known RTclients and the standard and user-defined message types in real time. Select one RTclient from the Clients list and select the Configure Watch Window... button or the Watch All Message Types button. You can also access the same window by highlighting a specific message type from the Select Message Type list and selecting the Watch Message Type button. Here is what you can monitor:

- to monitor messages received by a specific RTclient, select the RTclient from the Clients list in the Watch Messages Received window.
- to monitor all message types for that RTclient, select the Watch All Message Types button.
- to monitor specific message types for an RTclient, select the RTclient from the Clients list, highlight one or more message types from the Select Message Types list, and then press the Watch Message Type button.
- to monitor multiple RTclients, you must select one RTclient at a time. The system automatically polls that RTclient to retrieve the list of message types applicable to that RTclient. When polling is finished, the types of messages associated with that RTclient are displayed in the Select Message Type list.

The Watch Messages Received (client name) window appears when you select either the Configure Watch Window, Watch All Message Types, or Watch Message Type buttons.

Viewing Messages Received By An RTclient

In the Watch Messages Received window, after you select an RTclient and click on either the Configure Watch Window, Watch All Message Types, or Watch Message Type buttons, a Watch Messages Received (client name) window for that RTclient is shown. Multiple RTclient windows may be running simultaneously. Figure 39 shows the Watch Messages Received (client name) window. The Watch Messages Received (client name) window provides access to information about messages received by an RTclient.

Figure 39 Features of the Watch Messages Received Window

Note that you must suspend the queue (using the Online button) before you can configure the maximum number of messages in the queue.

Queue Mode

Seq. List Mode

Online Toggle

Watch Messages Received (/mon1)

Messages Received

Seq. Count	Queue Pos.	Message Type Name	Msg. Type #	Source
1	0	mon_client_msg_type_poll_resul	-450	/buyers
2	4	mon_client_time_poll_result	-444	/sellers
3	3	mon_client_general_poll_result	-442	/sellers
4	5	mon_client_option_poll_result	-448	/sellers
5	2	mon_client_general_poll_result	-442	/sellers
6	3	mon_client_time_poll_result	-444	/sellers
7	5	mon_client_option_poll_result	-448	/sellers
8	2	mon_client_general_poll_result	-442	/sellers
9	3	mon_client_time_poll_result	-444	/sellers
10	5	mon_client_option_poll_result	-448	/sellers

Show

☒ Queue

☐ Seq. List

Seq. List Max Size: <Unknown>

☐ Enable Log

Log File Filter

Log File...

View Log File

☐ Online

☐ Auto Run

Run 1

View Message

Close

Queue of Messages

Log File Control Buttons

Message Viewer Monitor and configure the information displayed in a specific message.

By default, the message types are displayed in real time (in Queue mode). Depending on the data rate of the project, messages may build and clear too quickly to view. When this happens, you need to change the default state of how information is displayed in order to view the information about the messages. The Seq. List mode displays messages in sequential order, allowing you to view information about each message type in the order in which it is received. In this sequential list mode, the list grows indefinitely unless you configure a maximum number of messages to be displayed.

To specify the maximum number of messages to be displayed:

1. Press the Online button to stop the monitoring of messages.
2. Select the Seq. List Max Size button. (The button itself displays the configured maximum numbered of messages if any, or <Unknown> is displayed if not set.)
3. Enter in the maximum number of messages to be displayed and press the Return key.
4. Press the Online button again to reactivate the monitoring of messages.

Saving Messages Received By An RTclient

You can save the messages in the queue to a log file, as well as display the log. You can also configure the type of information displayed in the log file using the Log File Filter button.

To start a log file:

1. Select the Online button to suspend the queue.
2. Select the Enable Log button.
3. Enter a log file name in the box.
4. Reactivate the queue by pressing the Online button again.

Viewing the Message Log File

Select the Online button to temporarily suspend the monitoring of messages, then select the View Log Message button to display the Message Log Viewer. You can view the log file through RTmon using the Message Log Viewer, as described, or through any text editor.

You can change the information filtered to the log file by changing the defaults in the Message Received Log Options window.

Changing What Is Filtered to the Log File

To change the type information filtered to the log file, select Log File Filter in the Watch Messages Received window. The Message Received Log Options window appears. By default, all of the options are enabled. The Message Received log options you can set are:

- Log Queue Info** Filters and displays the timestamp of the message, RTclient name, message type, queue position, and sequential counter.
- Log Messages** Filters and displays the body of the message. If disabled, only the log queue information is filtered to the log.
- Append Mode** When enabled, new messages are appended to the existing log.

If both the Log Queue and the Log Messages options are disabled, no information is filtered to the log.

Viewing Information About Individual Messages

To view detailed information about individual messages, select View Message in the Watch Messages Received window. The Message Viewer window appears. The information displayed in this window is message-specific.

When this window is open, you can highlight any of the messages listed in the Watch Messages Received window and the information specific to the highlighted message is displayed. Additionally, you can change the type of information displayed. For instance, if you do not want to view the message data, which can be very lengthy, toggle the Message Data option off. The settable parameters for this view are:

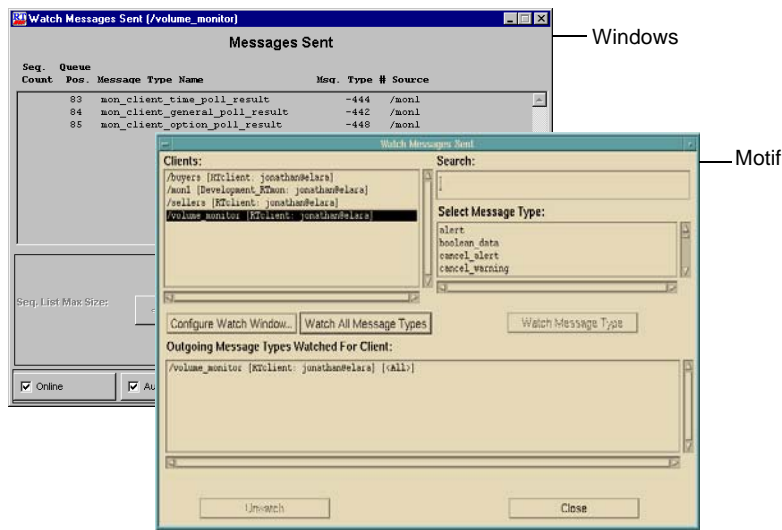
- Message Type** Displays the type of message being manipulated.
- Sender** Displays the name of the originator of the message.
- Destination** Displays the name of the subject to which the message is published.
- Maximum** Displays the maximum size of the data buffer.
- Size** Displays the size of the message being received.
- Current** Displays the current field of a message.
- Read Only** Displays whether or not the message can be modified.
- Priority** Displays the priority level of the message. The default priority level is zero. The range allowed is a 16-bit integer (-32768 to 32767).
- Delivery Mode** Displays the level of guarantee for a message when sent through a connection. The delivery mode options are best effort, some, and none. The default delivery mode is best effort.

LB Mode	Displays the mode for load balancing. The load balancing options are weighted, round_robin, sorted, and none. The default mode is none.
User Property	Displays a user-defined value that can be used for any purpose. This property is not used internally by SmartSockets.
Num Fields	Displays the number of fields that are included in the message data.
Message Data	Displays the payload of the message.

Monitoring Messages Being Sent

Select **Watch > Messages Sent...** from the Main menu to display the Watch Messages Sent window (shown in Figure 40).

Figure 40 Watch Messages Sent Window



This window displays the known RTclients and the standard and user-defined message types. Select one RTclient from the Clients list. Here is what you can monitor:

- to monitor messages sent from a specific RTclient, select the RTclient from the Clients list in the Watch Messages Sent window
- to monitor all the message types for that RTclient, select the Watch All Message Types button

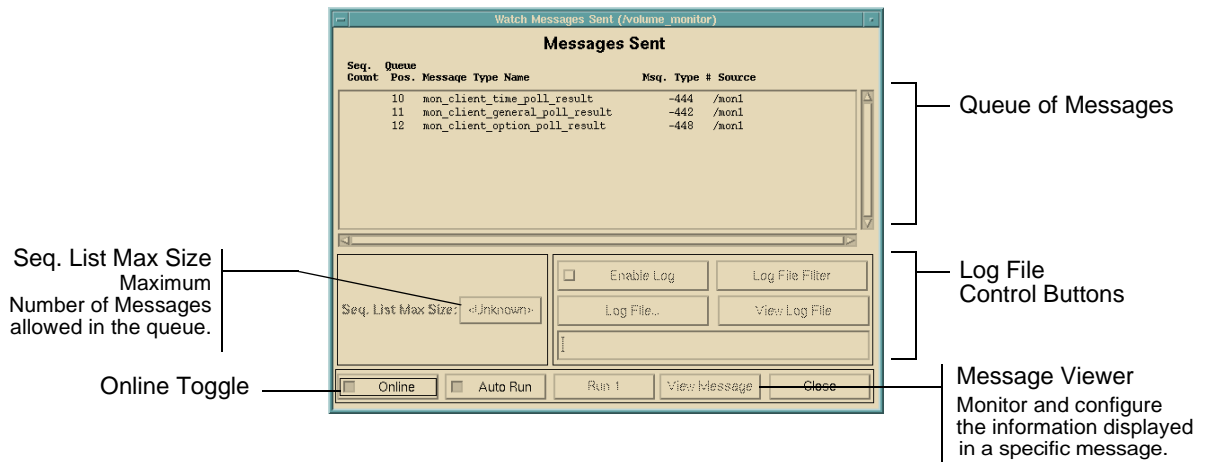
- to monitor specific message types for an RTclient, select the RTclient from the Clients list, highlight one or more message types from the Select Message Types list, and then press the Watch Message Type button
- to monitor multiple RTclients you must select one RTclient at a time. The system automatically polls that RTclient to retrieve the list of message types applicable to that RTclient. When polling is finished, the types of messages associated with that RTclient are displayed in the Select Message Type list.

The Watch Messages Sent (client name) child window, shown in Figure 41, appears when you select any one of the action buttons in the Watch Messages Sent window.

Viewing Messages Sent by an RTclient

In the Watch Messages Sent window, after you select an RTclient, either the Configure Watch window, Watch All Message Types, or Watch Message Type button must be pressed to open a Watch Messages Sent (client name) window for that RTclient. Multiple RTclient windows can be displayed simultaneously. Figure 41 shows the Watch Messages Sent (client name) window. The Watch Messages Sent (client name) window provides access to information about messages sent by an RTclient.

Figure 41 Features of the Watch Messages Sent Window



The messages appear in the list in sequential order in which they are sent. The list grows indefinitely unless you configure a maximum number of messages to be displayed.

To specify the maximum number of messages to be displayed:

1. Press the Online button to suspend the monitoring of messages.
2. Select the Seq. List Max Size button. (The button itself displays the configured maximum number of messages if any, or <Unknown> is displayed if not set.)
3. Enter in the maximum number of messages to be displayed and press the Return key.
4. Press the Online button again to reactivate the monitoring of messages.

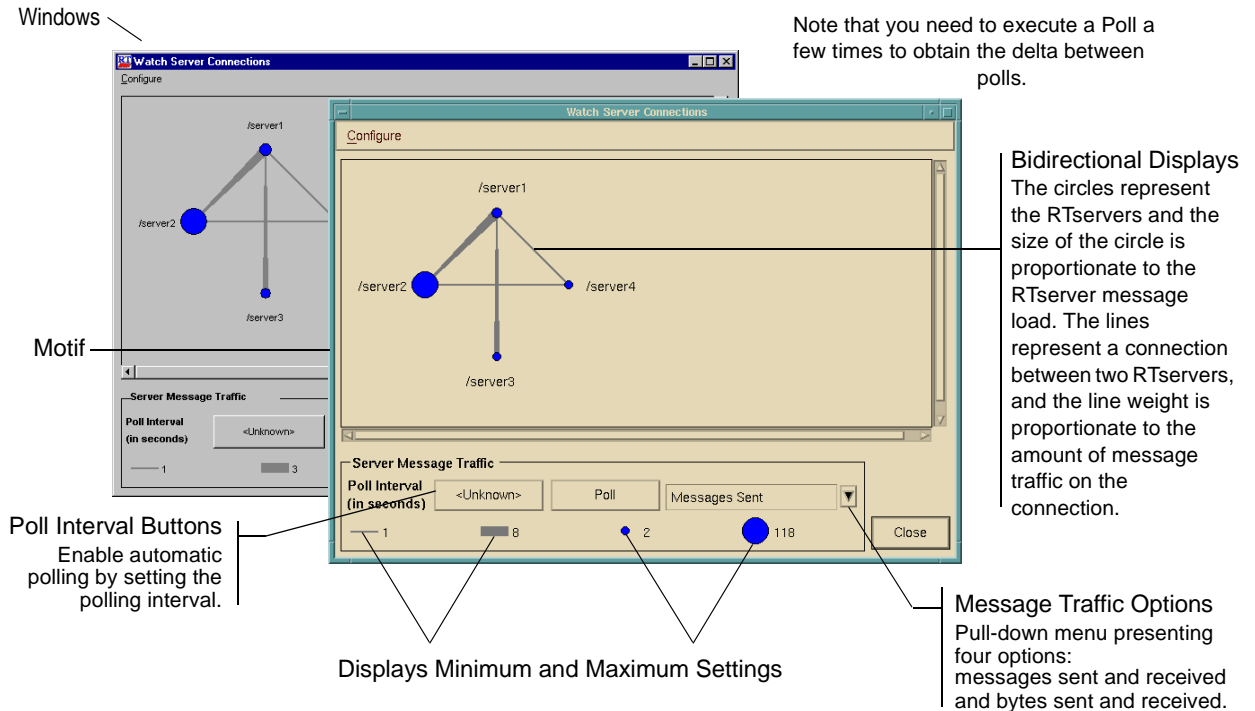
The Watch Messages Sent window is very similar to the Watch Messages Received window, and the logging functions and options are the same. See Viewing the Message Log File on page 458, Changing What Is Filtered to the Log File on page 459, and Viewing Information About Individual Messages on page 459.

Monitoring Server Connections

Select **Watch > Server Connections...** from the Main menu to display the Watch Server Connections window. The Watch Server Connections window presents an easy to read, real-time graphical display of information about the messages sent and received and the bytes sent and received from the RTserver.

The graphical displays present the delta between polls, as shown in Figure 42.

Figure 42 Watch Server Connections Graphical Chart

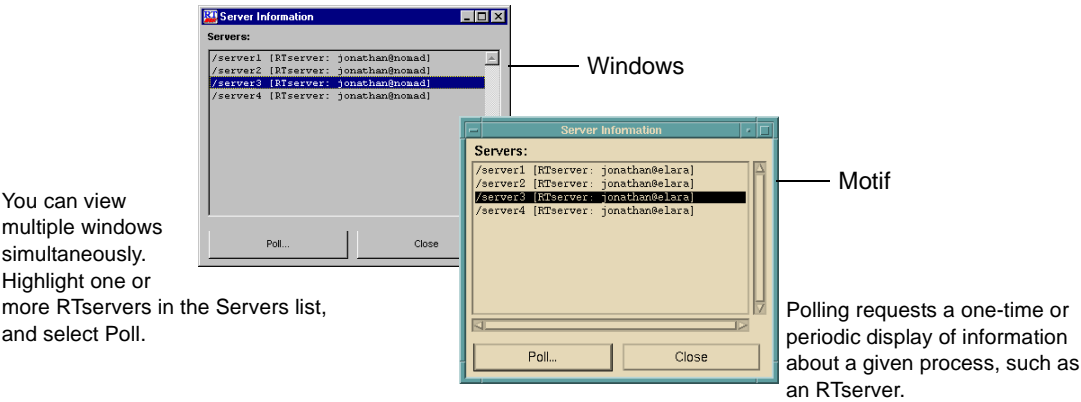


Select **Configure > Metrics...** from the Watch Server Connections window to display the Configure Metrics window. Use this window to configure the metric values used in the graphical displays. You can also use the Configure pull-down menu to scale the view to the current data.

Monitoring RTservers

Select **Poll > Server Information...** from the Main menu to display the Server Information window, shown in Figure 43. This window lists the RTservers in your project. You can poll and monitor information about each of the RTservers.

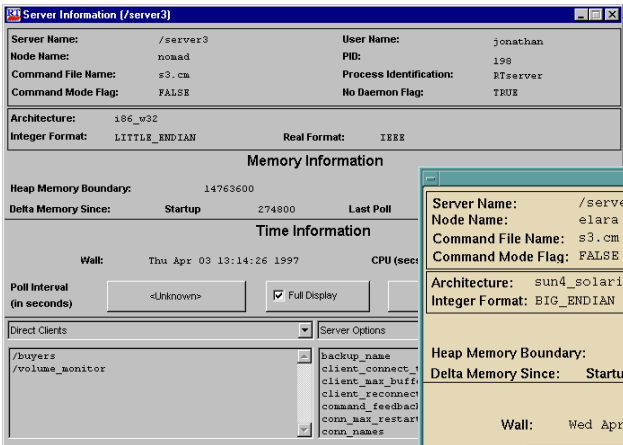
Figure 43 Server Information Window



To display detailed monitoring information about one or more of the RTservers listed in the window, highlight the RTserver(s), and press the Poll button. The Server Information (server name) window appears, as shown in Figure 44.

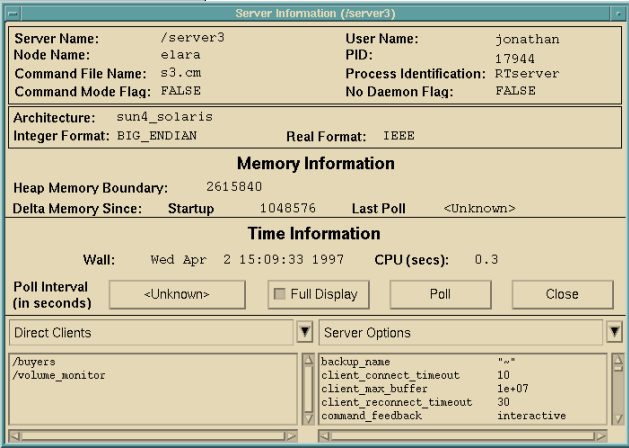
Figure 44 Server Information Window (Server Name)

Windows



These views are in Full Display mode. If you do not wish to view the window in full display mode, toggle the Full Display button.

Motif



There are four action buttons in the Server Information window: Poll Interval, Full Display, Poll, and Close.

This table presents a description of the information fields in the Server Information (server name) window when in full display mode. Note that those fields displayed only when in full display mode are noted with an asterisk.

Field Name	Description
Server Name	Displays the name of the RTserver.
Node Name	Displays the name of the node where the RTserver is running.
Command File Name	Displays the startup command filename for the RTserver.
User Name	Displays the name of the user who started the RTserver.
PID	Displays the process identification number of the RTserver.
Process Identification	Displays the process monitoring identification string.
No Daemon Flag	Displays the value of the <code>-no_daemon</code> command-line parameter. The options are true or false.

Field Name	Description
Architecture	Displays the type of hardware and operating system where the RTserver is running.
Integer Format	Displays the integer format used by the RTserver. The integer format options are big endian or little endian.
Real Format	Displays the real format used by the RTserver (such as IEEE or IBM 370).
Heap Memory Boundary	Displays the value of the edge for the RTserver virtual memory heap address space. This is used as a memory consumption gauge for a process.
Delta Memory Since Startup Last Poll	Displays the change in the heap memory boundary. Displays the changes since the RTserver started. Displays the changes from the point in time that the RTserver was last polled.
Wall	Displays the timestamp of the last poll.
CPU (secs)	Displays the CPU usage (in seconds) since the RTserver started.
Server Subscribes*	Displays the RTservers that are subscribed on behalf of the direct RTclients.
Client Subscribes*	Displays the union of the all the subjects to which all the direct RTclients are subscribed.
Direct Servers*	Displays the names of the RTservers connected to the RTserver that is being polled.
Direct Client Names*	Displays the names of the RTclients connected to the RTserver that is being polled.
Server Options*	Displays the names and values of the options available to the RTserver.

Checking Server Buffer Size

Select **Poll > Server Buffer...** from the Main menu to display the Server Buffer window, as shown in Figure 45. This window lists the RTservers in your project. To monitor buffer information for an RTserver, select one or more of the RTservers and click on the Poll button.

Figure 45 RTserver Buffer Windows

To display the buffer information about an RTserver, highlight the RTserver in the Servers list and press the Poll button.

Windows

Motif

Under normal circumstances, all the buffers shown in this window should be empty because an RTserver should rarely buffer data for any RTclients.

Polling requests a one-time or periodic display of information about a given process, such as an RTserver.

Connected	Process Name	Queue Count	# Bytes	Read Buffer	Write Buffer
/server2		0	0	0	0
/server3		0	0	0	0
/server4		0	0	0	0
/servers		0	0	0	0

Viewing RTclients for an RTserver

Select **Poll > Client Information...** to display the Client Information window. This window lists the RTclients in your project. There are four action buttons in the Client Information window:

- Poll Interval
- Full Display
- Poll
- Close

To display detailed monitoring information about the RTclients, highlight one or more of the RTclients in the Clients list and click on the Poll button. The Client Information (client name) window appears, which is very similar to the Server Information window shown in Figure 44.

The informational fields in the Client Information window when in full display mode are:

Field Name	Description
Client Name	Displays the name of the RTclient.
Node Name	Displays the name of the node where the RTclient is running.
Project	Displays the name of the current project.
User Name	Displays the name of the user who started the RTclient.
PID	Displays the process identification number of the RTclient.
Process Identification	Displays the process monitoring identification string.
Architecture	Displays the type of hardware and operating system where the RTclient is running.
Server Name	Displays the name of the RTserver to which the RTclient is connected.
Server Conn Name	Displays the logical connection name of the connection to the RTserver.

Field Name	Description
Architecture	Displays the type of hardware and operating system where the RTclient is running.
Heap Memory Boundary	Displays the value of the edge for the RTclient virtual memory heap address space. This is used as a memory consumption gauge for a process.
Delta Memory Since Startup Last Poll	<p>Displays the change in the heap memory boundary.</p> <p>Displays the changes since the RTclient started.</p> <p>Displays the changes from the point in time that the RTclient was last polled.</p>
Wall	Displays the timestamp of the last poll.
CPU (secs)	Displays the CPU usage (in seconds) since the RTclient started.
Subject Subscribes*	Displays the names of the subjects to which the RTclient is subscribed.
Options*	Displays the names and values of the options available to the RTclient.
Counted Licenses*	Displays the licenses currently being used.
Extra Licenses*	Displays the extra licenses currently being used, if any.



The asterisks mark the fields that are only displayed when in full display mode.

Viewing RTclient Message Traffic

Select **Poll > Client Message Traffic...** from the Main menu to display the Client Message Traffic window. This window displays the RTclients in your project and the associated incoming and outgoing message traffic for the specified RTclient. To monitor the traffic information, highlight one or more of the RTclients from the Clients list, and then press the Poll button. The RTclients appear in the Clients Polled area.

When a poll interval is configured using the Poll Interval button, the information displayed in this window is updated at the specified interval.

Viewing RTclient Message Traffic by Subject

Select **Poll > Client Subject Message Traffic...** from the Main menu to display the Client Subject Message Traffic window. This window displays the RTclients in your project and the associated incoming and outgoing message traffic for one or more of the client-subject pairs in your project. To monitor the traffic information, highlight one or more of the RTclients from the Clients list and one or more of the subjects from the Subjects list, and then press the Poll button. The client-subject pairs appear in the Clients Polled area.

To select an RTclient and all the subjects to which it is subscribing, highlight the RTclient from the Clients list, and then press on the Find Subjects button. Likewise, to select a subject and all the RTclients subscribed to the subject, highlight the subject from the Subjects list and press the Find Clients button.

To remove all poll information from the Client Subjects Polled area, press the Reset button.

Sending Messages with RTmon GDI

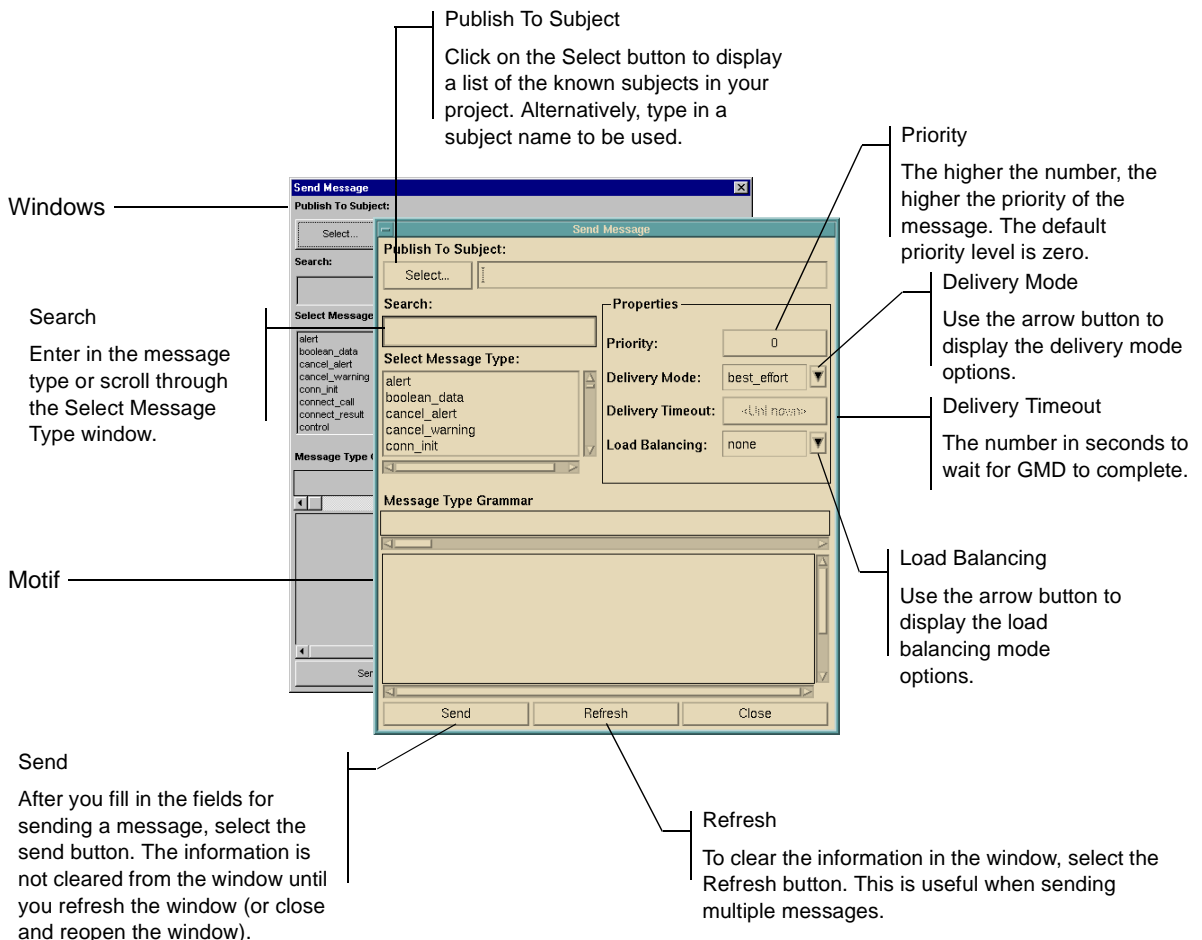


The RTmon GDI has been deprecated and may be removed in a future release.

To build and send a message to a subject, press the Send button from the Main RTmon window. The Send Message window appears.

There are many built-in attributes in the Send Message window. Figure 46 highlights the features built into this window.

Figure 46 Send Message Window



To send a message:

1. Specify a Subject. Only one subject at a time may be specified. When you press the Select button, the Subject Selection window appears. The subjects in your project are displayed in the Known Subjects list. Select one of the subjects displayed or type in the name of the subject in the Publish To Subject field.
2. Select a message type. The Search box speeds access to the Select Message Type list by scrolling to the place in the Message Type list that corresponds to the characters you type.

When a message type is selected and there is a grammar specified for that message type, it is displayed in the top line of the Message Type Grammar field. Additionally, one or more fields appear in the bottom portion of the window, prompting you for more information. Depending on the message type, the required information may be a simple text entry prompt, an editor of the appropriate type, or a list of legal values. The types of prompts and editors are defined in the table. If desired, set the priority of the message. The default is zero.

3. Optionally, specify a delivery mode. The options are Best_Effort (the default), Ordered, Some, and All. If Best_Effort is specified, a network failure might cause the message to be lost or delivered out of order. If Ordered is specified, a network failure might cause the message to be lost, but all delivered messages are in the order they were published. This is critical if some of the receiving clients are JMS clients because they cannot handle messages out of order. The modes Some or All trigger GMD messaging, which guarantees delivery to subscribers even during failures in addition to maintaining order during recovery. If Some is selected, it means that the message is guaranteed to be delivered to at least one subscriber. If All is selected, it means that the message is guaranteed to be delivered to all subscribers.
4. Optionally, set the delivery timeout of the message.
5. Optionally, specify the load balancing mode to be used to deliver the message. The load balancing options are None (the default), Round_Robin, Weighted, and Sorted.

For more information about sending messages, refer to Chapter 1, Messages.

The prompts and editors used when sending messages are:

Prompts	Editors
Simple Entry Prompt	<p>Most of the prompts for fields in the Send Message window are simple entry boxes (INT2, INT4, Identifier, REAL4 and REAL8). Enter the value and press the Return key.</p> <p>The value must be of the type specified in brackets above the box. If it is not, you are notified of an error.</p>
Group Entry Prompt	<p>Groups entry prompts ([Group]) have a slightly different function from Simple entry prompts. This entry prompt accepts an integer that specifies how many entry prompts to provide of the types listed inside the curly braces { 1..3 } in the Message Type Grammar field.</p> <p>When an integer is typed into this box and the Return key is pressed, that number of entry prompts appears below the group entry prompt.</p> <p>See the discussion of groups in the section Grammar on page 32.</p>
Message Entry Prompt	<p>Message entry prompts ([Message]) provide a Build Message window. This window is similar to the Send Message window, except that it does not provide as many sending options for the message.</p> <p>A message built with the Build Message window is a message within a message.</p>
String Entry Prompt	<p>String entry prompts ([String]) are similar to the simple entry prompt, with the addition of a More button, invoking a String Editor.</p> <p>The String Editor offers more capabilities than the string entry prompt. In the String Editor, you can:</p> <ul style="list-style-type: none"> • enter new line characters • view multiple lines of a string at once • scroll through long strings with scroll bars <p>Once a multi-line string has been entered using the String Editor, it is viewable in the string entry prompt by using the arrow keys.</p>

Prompts	Editors
Boolean Entry Prompt	Boolean entry prompts ([BOOL]) allow you to choose among the values Unknown, True, and False.
Array and Binary Entry Prompts (INT2, INT4, STR, REAL4, REAL8)	<p>If an entry prompt shows [INT2_ARRAY], [INT4_ARRAY], [STR_ARRAY], [REAL4_ARRAY], [REAL8_ARRAY], or [BINARY] as its type, left click on the box labeled <Empty> brings up an Item List Entry editor.</p> <p>This editor allows you to build lists of values of the particular type displayed on the entry prompt you selected:</p> <ul style="list-style-type: none">• To add a value to the list, left click on the Add button. An empty item box is displayed with the cursor already positioned in it. Enter the value and press the Return key.• To delete an item, left click on the Del button to the right of the item you want to delete. <p>For some types of items, an additional button labeled More... is associated with each item in the list. Left click on this button to bring up a String Editor. When the list is complete, left click on the OK button.</p>

Stopping RTmon GDI Processes

On UNIX and OpenVMS, when the RTmon GDI is started from a terminal emulator window, it is possible to stop it by typing Ctrl-c in the window where it was invoked. This method of exiting a process is mentioned here because sometimes during system development it is necessary to end a process that is in an infinite loop or otherwise stuck.

RTmon Command Interface

For more details on the commands available in RTmon, see Chapter 9 on page 583.

Starting a Command Interface Session

Type this command at the prompt to invoke the RTmon CI:

```
$ rtmon -runtime
```

A banner displays that shows the version and release of SmartSockets that you are running, and gives the contact information for TIBCO Product Support. After the banner, the RTmon prompt appears.

```
MON>
```



You can set the text that shows up as the prompt using the Prompt option. MON is the default prompt.

You can enter any RTmon commands at the prompt. The most frequently used commands are `watch`, `poll`, `send`, and `run`. After entering a `watch`, `poll`, or `send` command, you should follow it with a `run` command to ensure the command is executed:

```
MON> poll server_names
Polled for server_names.
MON> run
POLL> Current Servers:
/_workstation1.talarian.com_6848 [RTserver:
rtworks@workstation1.talarian.com]
/_workstation2_72522 [RTserver: RTWORKS@workstation2]
/_workstation3_11203 [RTserver: tom@workstation3]
```

The `POLL>` indicates RTmon is displaying the results of the poll. Type `quit` at the RTmon prompt to exit from the RTmon CI.

Chapter 7

Diagnosing Problems

It is often difficult to diagnose problems that are occurring within a distributed application. SmartSockets supplies you with a number of runtime debugging tools to aid in the debugging process. Some of the debugging tools available are:

- an RTmon process
- settable options within an RTclient or RTserver
- executable commands within an RTclient or RTserver
- callable API functions within an RTclient

This chapter describes how to detect and diagnose problems in the various components of a SmartSockets project. If your SmartSockets license includes multicast, check the special section, Multicast Troubleshooting on page 487.

Topics

- *Using RTmon, page 478*
- *Debugging Messages, page 478*
- *Diagnosing Connection Problems, page 479*
- *Diagnosing Memory Problems, page 480*
- *Diagnosing RTclient Problems, page 481*
- *Diagnosing RTserver Problems, page 484*
- *Multicast Troubleshooting, page 487*
- *Summary, page 491*

Using RTmon

The primary tool to use in debugging a SmartSockets project is RTmon. RTmon is a stand-alone standard RTclient built specifically for monitoring and debugging distributed applications that use RTservers for publish-subscribe communications. RTmon cannot be used to monitor and debug peer-to-peer connections.

Debugging Messages

One of the easiest ways to debug message-related problems is to write messages to a message file. Message files do not contain all properties of a message (such as the priority and sender properties), but message files are relatively compact and easy to read.

In RTclients and RTservers, several options are available to easily start or stop logging messages to message files. Use the `TipcMsgFileWrite` function in any code to write a message to a message file. RTmon can also read messages being sent to an RTclient, as well as log them into a message file.

If you need more precise details about messages or if message files are not working, use the `TipcMsgPrint` function to print all the information about a message. This is useful for validating that all the properties of a message have been set correctly and that the message data uses the expected field types and field values. The option `Real_Number_Format` controls the level of precision SmartSockets uses to print real numbers, including the real number fields in messages.

Debugging Message Types and Message Files

Messages cannot be written to message files if the field types in the message do not exactly match the grammar listed in the message type (unless the verbose format is used in the grammar when the message type is created). In this situation, use the `TipcMtPrint` and `TipcMsgPrint` functions to print and compare message type information to message field types, and resolve the field type mismatches.

Diagnosing Connection Problems

The most common connection problems are usually related to messages, and the methods in the previous section can be used. However, connection-related problems can also involve the improper use of callbacks. As discussed earlier, RTmon cannot be used to debug direct peer-to-peer connections that do not use RTserver.

Receiving Unwanted Messages

When a message is processed with `TipcConnMsgProcess` and there are no connection process callbacks for messages of that type and no connection default callbacks, `TipcConnMsgProcess` silently does nothing with the message. This is called an unwanted message. Receiving a large number of unwanted messages can noticeably slow processes. It is always a good practice to use `TipcConnDefaultCbCreate` to create a connection default callback that calls `TipcMsgPrintError` for unwanted messages. The output from `TipcMsgPrintError` is similar to:

```
WARNING: Received unwanted message at time 19.
type = numeric_data
sender = </_workstation.talarian.com_5031>
sending server = </_workstation.talarian.com_4982>
dest = </eps>
max = 2048
size = 64
current = 0
read_only = false
priority = 0
delivery_mode = best_effort
ref_count = 1
seq_num = 0
resend_mode = false
user_prop = 0
data (num_fields = 4):
    str "y"
    real8 2
    str "z"
    real8 3
```

Diagnosing Memory Problems

One of the following will occur if an application is unable to allocate enough memory to satisfy a request:

If the application runs out of memory while it is....	The result is....
reading a message	the connection is dropped
decoding a message	the message is dropped
sending a message	the message is dropped

Certain messages, such as protocol messages and GMD messages, will trigger a call to `abort()` (the application will core) when the available memory is exhausted. The only exception is when decoding, in which case the failure is promoted from dropping the message to dropping the connection. Should this occur, a warning with diagnostic information is printed to the location specified by the `trace_file` option (stdout by default).

Diagnosing RTclient Problems

RTclient can use API functions to help diagnose problems. Options and commands can also be used to debug RTclient in many situations without requiring any programming. RTmon should be used whenever possible to help pinpoint and debug problems within RTclients.

Connections and Messages

RTclient uses connections and messages, and you need the techniques described in the previous sections. Always use `TipcSrvDefaultCbCreate` to create a default callback in the connection to RTserver. The callback function then calls `TipcMsgPrintError` to immediately expose unwanted messages. The function `TipcSrvPrint` can be used to print all the information about the connection to RTserver.

Why RTclient Is Not Receiving Data

During the development of a project with RTclient, it can be puzzling why an RTclient process does not appear to be receiving data. There are several reasons why this might occur, including:

- the `Server_Names` option of RTclient is not the same as that for the other RTclient processes, or the RTclient processes are using different groups of RTserver processes
- the `Project` option of RTclient is not the same as that for the other RTclient processes. If this happens, RTserver does not create the logical link needed so that messages flow between RTclient and the rest of the project.
- no data is really being sent, or the sending RTclient process has the option `Server_Msg_Send` set to `FALSE`
- the sending RTclient process is using message type numbers that are different from the message type numbers being used by the receiving RTclient processes
- the sending RTclient process is sending the message to the wrong subject
- the sending RTclient has aborted for some reason
- the sending or receiving RTclient or RTserver is out of memory

Tracing Lost Messages

Message files can often be used to incrementally track down why data is not being received. You can start at either the sending or receiving RTclient process and work through the connections to find the problem. The RTclient and RTserver logging options can be used to trap the flow of data at various points.

For example, if a subscribing RTclient does not appear to be receiving any messages from a publishing RTclient, try setting the option `Log_In_Data` in the subscribing RTclient to see if it is really receiving the messages. If the messages appear in the resulting message file, then the problem lies in that subscribing RTclient. It might not be creating the proper connection process callbacks.

If the messages do not appear in the resulting message file, try setting the option `Log_Out_Data` in the publishing RTclient to determine if it is really sending the messages.

If the problem appears to lie in RTserver, you can log messages in RTserver to debug the problem. See Useful Commands on page 483 for more information on RTserver debugging commands.

Useful Options

These RTclient options can be used for debugging:

<code>Command_Feedback</code>	have RTclient provide feedback on commands executed.
<code>Log_In_Data</code>	file to write incoming data-related messages.
<code>Log_In_Internal</code>	file to write incoming internal messages.
<code>Log_In_Status</code>	file to write incoming status messages.
<code>Log_Out_Data</code>	file to write outgoing data-related messages.
<code>Log_Out_Internal</code>	file to write outgoing internal messages.
<code>Log_Out_Status</code>	file to write outgoing status messages.

The `Log_*` options allow you to easily write messages into message files when entering or leaving RTclient.

Useful Commands

These RTclient commands are provided to make debugging easier:

- `setopt` sets or displays the value of an option.
- `stats` returns information on memory usage, elapsed wall clock, and CPU time.
- `subscribe` lists the subjects being subscribed to.

The `setopt` command (when issued without any arguments) prints the current setting of all RTclient options.

The `stats` command provides information about the CPU usage of RTclient. The information includes the total amount of CPU and wall clock time since RTclient was started, the amount of memory the program is using (`sbrk` address), and the differences since the last `stats` command was issued. Below is an example of the `stats` command:

```
CLIENT> stats
Total accumulated CPU time: 3.866 seconds
Total frames processed: 12
Current sbrk address: 251840
Differences since last stats command:
  CPU time, 0.383 seconds, wall time, 11.174 seconds
  Frame count: 5, Frame rate: 0.447 frames per second
  Sbrk address changed by 0 bytes.
CLIENT>
```

The `subscribe` command (when issued without any arguments) prints a list of the subjects to which RTclient is currently subscribing.

Diagnosing RTserver Problems

The default for starting RTserver is to start the optimized version, in which validations and checking are turned off. Your first step in diagnosing RTserver problems is to ensure you are running the check version of RTserver so you can collect more information about whatever problem you are having. Start RTserver using:

```
rtserver -check
```

This starts the check (also called debug) version of RTserver.

Because RTserver does not have any API functions, no C/C++ code can be added to RTserver to help diagnose problems. There are several files, command-line arguments, options, and commands available to debug RTserver without requiring any programming. Also, RTmon can be used to poll or watch information in RTserver. These and other debugging features are described below.

Files Created by RTserver

When RTserver starts, it creates a debug file as specified by the `-trace_file` argument specified on the `rtserver` command that you used when you started the RTserver. If you did not specify the `-trace_file` argument, the default is to use standard output (`stdout`), which is printed to the console.

If RTserver crashes or is having problems, the debug file (if you specified one) is the first place to look.

Each time RTserver prints something to its debug file, it immediately flushes the data to disk so that it can be easily typed out. When RTserver exits cleanly, it checks the size of the debug file and automatically removes the debug file if the file is empty. This automatic cleanup prevents hundreds or even thousands of useless debug files from clogging the filesystem that contains the directory where the debug files are written. The size of the debug file can be regulated with the `Trace_File_Size` option. If RTserver crashes, this debug file is a good place to look for error messages. The operating system modification date of the debug file can be used to correlate debug files with previous RTserver sessions.

On OpenVMS, RTserver also creates these files when RTserver starts a detached background process:

- `stdout` is stored in `SYS$SCRATCH:RTSERVER_OUT_Node_User_Counter.TMP`.
- `stderr` is stored in `SYS$SCRATCH:RTSERVER_ERR_Node_User_Counter.TMP`. This file is not created unless RTserver writes something to `stderr`.
- a small command procedure is stored in the file `SYS$SCRATCH:RUN_RTSERVER_Node_User_Counter.TMP`.
- an empty lock file, `SYS$SCRATCH:RTSERVER_Node_Pid_OK.TMP`. RTserver waits for the detached background process to create this file once it has finished initialization.

For OpenVMS:

Node is the network node name of the computer on which RTserver is running.

User is the user name of the account starting RTserver.

Counter is a small number, used to generate the OpenVMS process name of the RTserver process.

Pid is the operating system process identifier of the RTserver.

If RTserver fails to start successfully on OpenVMS, you can look at these files for information such as stack traces and error messages.

Useful Command-Line Arguments

The most useful RTserver command-line arguments for debugging are:

- `-trace_file`, which allows you to specify a file name for the debug file and have the debug file saved rather than printed to the console
- `-trace_level`, which allows you to specify what amount of information gets written to the debug file

There are many different levels of trace information you can have written to the debug file. The most detailed setting is `debug`. For complete information on the settings for trace level, see *Starting RTserver* on page 288.

Useful Options

These RTserver options are useful for debugging:

- `Command_Feedback` — have RTserver provide feedback on commands executed.
- `Log_In_Client` — file to log incoming messages from RTclient processes.
- `Log_Out_Client` — file to log outgoing messages to RTclient processes.
- `Log_In_Server` — file to log incoming messages from other RTserver processes.
- `Log_Out_Server` — file to log outgoing messages to other RTserver processes.
- `Trace_File` — specify name and location of the trace file.
- `Trace_Flags` — format of the information in the trace file.
- `Trace_Level` — specify level of detail for the trace information being put in the trace file.
- `Verbose` — have RTserver output debugging information.

The `Log_*` options allow you to easily log messages into message files entering or leaving RTserver. The `Trace_Level` and `Verbose` options provide a way to watch the detailed operations of RTserver. When the `Verbose` option is set to `TRUE`, RTserver prints out much information to the debug file as new processes connect, existing processes disconnect, and subjects are operated on. Setting `Verbose` to `TRUE` is equivalent in level of detail to setting `Trace_Level` to `verbose`.

Multicast Troubleshooting

This section provides several general multicast troubleshooting tips to help you deploy your multicast applications. These tips are not applicable to unicast systems.

Verify Your Configuration

Before trying to resolve a complicated multicasting issue, follow these tips to eliminate or avoid basic problems:

- Become familiar with the operation of your multicast application on a simple network before trying to make it work on a complicated one.
- The "bottom-up" approach is generally best. First focus on getting the lowest layers of the network stack working.
- Verify that the network has good unicast connectivity between the sender and all receivers before addressing multicast connectivity problems.
- Ensure that multicast has been enabled across the test or production environment by checking that the switches and routers are multicast-enabled.
- From a network point of view, try connecting your sending and receiving hosts to the same hub, not a switch or a router, and confirm that you have multicast connectivity. Once that works, move on to more complicated multicast networks.
- From a SmartSockets point of view, try testing with two receiving clients that are on the same subnet as your RTgms process. Once that works, expand the test to other subnets.

Verify Your PGM Option Settings

Some common problems are caused because the options affecting PGM configuration are not set correctly. Check the value of an option by looking at how it is set in the applicable command (.cm) file.

Group_Names Option

Set in the `$RTHOME/standard/rtgms.cm` file for your RTgms process.

Check that the receiving RTclients and the RTgms to which they are connecting use the same value for Group_Names. For more information on setting options for RTclients, see RTclient Options Summary on page 501. For more information on the option, see Group_Names on page 537.

Pgm_Udp_Encapsulation Option

Set in the `$RTHOME/standard/mcast.cm` file for your RTgms and RTclient processes.

Check that the receiving RTclients and the RTgms to which they are connecting use the same value for Pgm_Udp_Encapsulation. If the value is not present in those files, the default value is used and does match correctly.

If Pgm_Udp_Encapsulation is set to 0 and your operating system is UNIX, all receiving RTclients and the RTgms processes must be run as root. If Pgm_Udp_Encapsulation is set to 0 and your operating system is Windows 2000, all receiving RTclients and the RTgms processes must have administrator authority.

If Pgm_Udp_Encapsulation is set to 1 to use UDP encapsulation, no PGM subscribers can be run on the machine where RTgms is running. This is because NAKs from PGM might not be correctly processed by the UDP protocol when RTgms and a PGM subscriber are running on the same machine. This problem does not occur when PGM is using raw IP sockets (Pgm_Udp_Encapsulation set to 0).

For more information on the option, see Pgm_Udp_Encapsulation on page 667.

Pgm_Source_Group_Ttl and Pgm_Receive_Nak_Ttl Options

Set in the `$RTHOME/standard/mcast.cm` file for your RTgms and RTclient processes.

These options must be set to the number of router hops a packet of data takes. The number is calculated starting from the source, the RTgms, and ends with the subscriber, the receiving RTclient. The default value of these options is 1, which does not allow the packet to go beyond the first hop, either a router or switch. For more information on these options, see Pgm_Source_Group_Ttl on page 663 and Pgm_Receive_Nak_Ttl on page 660.

Tracing Problems to Their Source

Once you eliminate simple problems, tracing the source of a problem can be complicated in a multicast application. Here are some useful tips:

- Make sure that multicast streams are being generated with a TTL adequate to reach their destination via the longest-possible path through the network. See *Troubleshooting Multicast Problems with Cisco Systems Routers* on page 490 to check whether the TTL is too low.
- Start at the source and trace your way through each switch and router to all receivers.
- Use the SmartSockets PGM `pthrpt` command on the source to generate test streams. For example:

```
pthrpt -t 10 -r 10000 -m 224.13.13.13
```


generates a 10 Kbps stream with a TTL of 10 to group address 224.13.13.13. You can ask TIBCO Product Support for more information about PGM commands or how to get the *SmartPGM User's Guide*.
- Use the SmartSockets PGM `pbw` command or Cisco Systems IOS `show ip mroute active` command for monitoring the presence of test streams. See *Troubleshooting Multicast Problems with Cisco Systems Routers* on page 490.
- Test receiver to last-hop router communication by running the SmartSockets Multicast `pthrpt` command on the receiver.
- On UNIX, use the `tcpdump` command or the Solaris `snoop` command to monitor test streams.
- If you see packet loss as multicast rates go up, look for routers or switches that are configured to limit the broadcast rate. These generally also limit the multicast rate. For example, Cisco Systems Catalyst 5000 series switches can be configured to limit the packet for each second or percentage of broadcast and multicast traffic with the `set port broadcast` command.
- Try using UDP encapsulation to see if the network will pass UDP even if it is not passing PGM. This behavior has been seen in some networks with Catalyst 4000 series switches. See *UDP Encapsulation of PGM* on page 678.

Troubleshooting Multicast Problems with Cisco Systems Routers

A Cisco Systems router might refuse to forward multicast packets because their TTL (Time To Live) is too small. Use the IOS command `show ip traffic` and watch the number of bad hop count packets shown on the second line of output. This counter increments every time IOS throws a packet away because its TTL was too small. For example:

```
Router> show ip traffic
IP statistics:
  Rcvd: 322205025 total, 8805271 local destination
        2 format errors, 0 checksum errors, 185555472 bad hop
count
```

This shows that 59% ($185555472 / (322205025 - 8805271)$) of all traffic that could have been forwarded was not because the TTL was too small.

The IOS command `show ip mroute active` is a quick way to see the most active multicast traffic passing through a router. For example:

```
Router> show ip mroute active
Active IP Multicast Sources - sending >= 4 kbps

Group: 224.13.13.13, (?)
Source: 10.168.4.5 (Lisle.Stress1.Talarian.Com)
Rate: 3 pps/12 kbps(1sec), 9 kbps(last 25 secs), 10 kbps(life
avg)
```

The IOS command `show ip mroute count` is often helpful in determining if a router can see a multicast stream, how fast the stream is going, if the stream is being forwarded, and if not, why not. For example:

```
Router> show ip mroute count
IP Multicast Statistics
8 routes using 4024 bytes of memory
6 groups, 0.33 average sources per group
Forwarding Counts: Pkt Count/Pkts per second/Avg Pkt Size/Kilobits
per second
Other counts: Total/RPF failed/Other drops(OIF-null, rate-limit
etc)

Group: 225.0.11.66, Source count: 0, Group pkt count: 0
Group: 225.0.11.68, Source count: 0, Group pkt count: 0
Group: 224.0.1.40, Source count: 0, Group pkt count: 0
Group: 224.0.1.1, Source count: 1, Group pkt count: 1
Source: 216.0.13.9/32, Forwarding: 1/0/76/0, Other: 2/1/0
Group: 225.0.11.11, Source count: 0, Group pkt count: 0
Group: 224.13.13.13, Source count: 1, Group pkt count: 108
Source: 10.168.4.5/32, Forwarding: 108/3/537/12, Other: 108/0/0
```

Note the last group. It shows that the router is seeing 3 packets for each second and 12 Kbps to the group 224.13.13.13 from source 10.168.4.5.

See the Cisco Systems web site for excellent starting points for additional multicast troubleshooting information:

- Basic Multicast Troubleshooting Tools page covers the IOS commands that are most useful in troubleshooting multicast problems. Because this page also discusses host commands such as `netstat`, it might be of interest even if you do not have Cisco Systems routers.
- IP Multicast Troubleshooting Guide contains links to several case studies that illustrate use of the above tools and appropriate troubleshooting techniques.

Multicast Testing Tools

These tools can be useful in troubleshooting and monitoring multicast networks:

- Ethereal (for UNIX)
- Tcpdump (for UNIX)
- WinDump (Tcpdump for Windows)
- NLANR Multicast Beacon
- MBone tools for Windows

Summary

SmartSockets provides many tools that allow you to better understand the internal processing as data is received, processed, and sent out to other RTclient processes. You can inspect the major components of messages, connections, RTclient and RTserver, look at the CPU usage, and even control the RTclient and RTserver execution. These tools make it much easier for you to debug projects that are built using SmartSockets.

Chapter 8 Options Reference

This chapter describes the options available to RT processes such as RTservers and RTclients. You can set these options in various ways, depending on the option and the type of RT process. For information on options that apply only to RTgms, see Chapter 10, Using Multicast.

Topics

- *Setting Option Values, page 494*
- *Startup Command Files, page 498*
- *RTclient Options Summary, page 501*
- *RTserver Options Summary, page 505*
- *RTmon Options Summary, page 509*
- *Multi-Thread Mode, page 512*
- *Option Reference, page 514*

Setting Option Values

The most common way to set an option value is by specifying the `setopt` command with the option name and value in the startup command file for the RT process. Every RT process searches for command files upon startup, and executes the commands in those files. If you want to set the value for a named option, use the `setnopt` command with the option name and value in the startup command file. See [Startup Command Files](#) on page 498.

You can also use a CONTROL message. CONTROL messages are messages of type CONTROL and can contain any valid command. To change an option value, use a CONTROL message that contains the `setopt` or `setnopt` command. CONTROL messages can be used for any RT process. The examples in [Working With RTclient](#) on page 175 show how to compose and send CONTROL messages. CONTROL messages received by an RT process are logged to the RT processes trace file if enabled.

You are not required to set any option values. All required options have default values that are used at startup. However, tailoring the option values to optimize your SmartSockets system is key to unleashing the power and flexibility of this messaging system.

For information on setting options for RTgms, see [RTgms Options](#) on page 650.

RTclient Options

RTclient options can be set to specific values by defining them in a command file, by calling the API function `TutCommandParseStr`, or by calling one of the `TutOptionSetType` API functions. You can also send a CONTROL message to a subject to which the RTclient is subscribed.

RTserver Options

RTserver options can be set to specific values by defining them in the `rtserver.cm` command file. Option values that have been specified in the command file are set each time RTserver is started, or they can be modified using the `setopt` command in a CONTROL message.

Certain options can be modified dynamically for a particular RTserver connection using an ADMIN_SET message. When you send the message to a particular RTserver process, the options apply only to outbound data sent on the specified connection. The connection can be:

- a connection between an RTserver and an RTclient
- a connection between two RTservers
- a group channel between an RTserver and an RTgms

The options that can be set this way are for network bandwidth rate control. For more information on bandwidth rate control, see Controlling Network Bandwidth and Usage on page 305.

The ADMIN_SET message used for RTservers is:

```
T_MT_ADMIN_SET_OUTBOUND_RATE_PARAMS
T_STR      connection
T_INT4     token_rate
T_INT4     max_tokens
T_REAL8    burst_interval
```

where:

connection for an RTclient or RTserver connection, *connection* is the unique subject name of the RTclient or RTserver for which you want these options set, and for an RTgms connection, *connection* must be the multicast group name of the connection, matching an existing group name specified by the Group_Names option.

token_rate is the rate, in bytes a second, at which tokens accumulate. A value of -1 indicates no change.

If the value you specified for *connection* is the unique subject name of an RTclient, *token_rate* is equivalent to the Client_Token_Rate option.

If the value you specified for *connection* is the unique subject name for an RTserver, *token_rate* is equivalent to the Server_Token_Rate option.

If the value you specified for *connection* is the group name for a connection to an RTgms, *token_rate* is equivalent to the Group_Token_Rate option.

max_tokens is the maximum number of tokens that can accumulate. A value of -1 indicates no change.

If the value you specified for *connection* is the unique subject name of an RTclient, *max_tokens* is equivalent to the Client_Max_Tokens option.

If the value you specified for *connection* is the unique subject name for an RTserver, *max_tokens* is equivalent to the Server_Max_Tokens option.

If the value you specified for *connection* is the group name for a connection to an RTgms, *max_tokens* is equivalent to the Group_Max_Tokens option.

burst_interval is the burst interval in number of seconds. A value of -1.0 indicates no change.

If the value you specified for *connection* is the unique subject name of an RTclient, *burst_interval* is equivalent to the Client_Burst_Interval option.

If the value you specified for *connection* is the unique subject name for an RTserver, *burst_interval* is equivalent to the Server_Burst_Interval option.

If the value you specified for *connection* is the group name for a connection to an RTgms, *burst_interval* is equivalent to the Group_Burst_Interval option.

RTmon Options

RTmon options can be set to specific values by defining them in the `rtmon.cm` command file. Option values that have been specified in the command file are set each time the RTmon is started, or they can be modified from the RTmon command interface using the `setopt` command.

Specifying Options

Option names are shown in mixed case in this reference to differentiate them from commands, but options are not case sensitive.

When entering multiple values for list options, they must be separated by commas. Values for options of types String or String List can be entered with or without double quotes, with these exceptions:

- double quotes must be used for values that include a space, tab, comma, or semicolon
- quotes must not be used for keyword values, such as `_all`

You can use the standard C comment indicator `//` to comment out a line in a command file. For example:

```
setopt project          ST1A
setopt server_names     workstation1
//setopt server_disconnect_mode  gmd_failure
setopt server_disconnect_mode  warm
```

In this case, the value `warm` is used for `Server_Disconnect_Mode`.

For Java, instead of C, see the *TIBCO SmartSockets Java Library User's Guide and Tutorial*.

Startup Command Files

RTclient

An RTclient process does not have any standard startup command files. It is up to you to decide whether command files should be loaded, and to create and define those files. Startup command files can be loaded with the `TutCommandParseFile` function. For information on creating and using RTclient command files, see the *TIBCO SmartSockets Tutorial*.

This example illustrates a typical RTclient command file:

```
setopt project           HST
setopt server_names      workstation1
```

RTserver

An RTserver process has a standard startup command file, named `rtserver.cm`. The RTserver startup command files contain generic information that RTserver needs to know, such as how to locate and connect with RTclient processes, how to locate and connect with other RTserver processes, and how often to check for network failures.

This example illustrates a typical RTserver startup command file:

```
setopt editor             emacs
setopt real_number_format %f
setopt server_read_timeout 10
```

RTserver recognizes three levels of startup command files. When first invoked, it searches for and executes the commands in each file, in this order:

1. the system-level `rtserver.cm` file in the SmartSockets `standard` directory

RTserver searches for a system-level process command file `rtserver.cm` in the SmartSockets directory `standard`, in `RTHOME`:

- **UNIX:** `$RTHOME/standard`
- **OpenVMS:** `RTHOME:[STANDARD]`
- **Windows:** `%RTHOME%\standard`

The `rtserver.cm` file can be modified to reflect RTserver options that are meant to be system-wide. To edit this file, change the current directory to the SmartSockets `standard` directory and use an editor to add or change the system-wide option settings.

2. the user-level `rtserver.cm` file in the user's home directory

RTserver searches for an `rtserver.cm` file in the user's home directory and, if found, executes the commands in that file. This file is the ideal place to set options specific to all your projects. To create this file, use an editor to open a new file named `rtserver.cm` in your home directory and add the options you choose.

The location of the home directory that RTserver searches varies by operating system:

- **UNIX:** `$HOME`
- **OpenVMS:** `SYS$LOGIN`
- **Windows:** `%HOME%`

3. the file specified by the `-command` argument, or the local-level `rtserver.cm` file in the current directory (if the `-command` argument is not specified)

RTserver reads and executes the `rtserver.cm` file found in the current directory, that is, the directory from which RTserver is being run. Use this file for any application-specific option declarations. The local command file is read last, allowing you to override any values set for RTserver options in other command files.



If you have configured Basic Security features (as outlined in Security on page 275), RTserver also loads the command file `sdbasic.cm`. To locate this file, RTserver uses the same three discovery techniques outlined here for locating `rtserver.cm`.

RTmon

An RTmon process has a standard startup command file, named `rtmon.cm`. The RTmon startup command files contain generic information that RTmon needs to know, such as what values to use for IPC-related timeouts, what project to monitor, and what node RTserver resides on.

This example illustrates a typical RTmon startup command file:

```
setopt project          HST
setopt server_names    cosmos
setopt editor          emacs
```

RTmon recognizes three levels of startup command files. When first invoked, it searches for and executes the commands in each file, in this order:

1. the system-level `rtmon.cm` file in the SmartSockets `standard` directory

RTmon searches for a system-level process command file `rtmon.cm` in the SmartSockets directory `standard`, in `RTHOME`:

- **UNIX:** `$RTHOME/standard`
- **OpenVMS:** `RTHOME:[STANDARD]`
- **Windows:** `%RTHOME%\standard`

This file can be modified to reflect RTmon options that are meant to be system-wide. To edit this file, change to the directory and use an editor to add or change the system-wide option settings.

2. the user-level `rtmon.cm` file in the user's home directory

RTmon searches for an `rtmon.cm` file in the user's home directory (specified by the `HOME` environment variable) and, if found, executes the commands in that file. This file is the ideal place to set options specific to all your projects. To create this file, use an editor to open a new file named `rtmon.cm` in your home directory:

- **UNIX:** `$HOME`
- **OpenVMS:** `SYS$LOGIN`
- **Windows:** `%HOME%`

and use the editor to add the options you choose.

3. the file specified by the `-command` argument, or the local-level `rtmon.cm` file in the current directory (if the `-command` argument is not specified)

RTmon reads and executes the `rtmon.cm` file found in the current directory, that is, the directory from which RTmon is being run. It is in this file that you place any project-specific option declarations, such as the name of the project and where to find RTserver. The local command file is read last, allowing you to override any values set for RTmon options in other command files.

RTclient Options Summary

The table summarizes the relevant options available in all RTclient processes. All of these options can be modified using the `setopt` command from the RTclient command interface.

Table 15 RTclient Options

Option Name	Type	Default
Auth_Data_File	String	None
Backup_Name	String	UNIX: ~ OpenVMS: None Windows: ~
Catalog_File	String	\$RTHOME/standard/tal_ss.cat
Catalog_Flags	String List	id
Command_Feedback	Identifier	interactive
Compression	Boolean	FALSE
Compression_Args	String	6
Compression_Name	String	zlib
Compression_Stats	Boolean	FALSE
Default_Msg_Priority	Numeric	0
Default_Protocols	Identifier List	UNIX: local, tcp OpenVMS: tcp Windows: tcp
Default_Subject_Prefix	String	None
Editor	String	UNIX: vi OpenVMS: edt Windows: notepad

Table 15 RTclient Options (Cont'd)

Option Name	Type	Default
Enable_Control_Msgs	String List	echo,quit
Group_Names	String List	rtworks
Ipc_Gmd_Auto_Ack	Boolean	TRUE
Ipc_Gmd_Auto_Ack_Policy	String	first_destroy
Ipc_Gmd_Directory	String	UNIX: /tmp/rtworks OpenVMS: sys\$scratch Windows: %TEMP%\rtworks
Ipc_Gmd_Type	String	default, which means file-based GMD is attempted
Log_In_Data	String	None
Log_In_Internal	String	None
Log_In_Msgs	String	None
Log_In_Status	String	None
Log_Out_Data	String	None
Log_Out_Internal	String	None
Log_Out_Msgs	String	None
Log_Out_Status	String	None
Monitor_Ident	String	RTclient
Monitor_Scope	String	/*
Pgm_*		For information on these options, see Chapter 10, Using Multicast.
Project	Identifier	rtworks
Proxy_Password	String	UNKNOWN
Proxy_Username	String	UNKNOWN

Table 15 RTclient Options (Cont'd)

Option Name	Type	Default
Real_Number_Format	String	%g
Sender_Get_Reply	Boolean	FALSE
Server_Async_Subscribe	Boolean	TRUE
Server_Auto_Connect	Boolean	TRUE
Server_Auto_Flush_Size	Numeric	8192
Server_Delivery_Timeout	Numeric	30.0
Server_Disconnect_Mode	Identifier	gmd_failure
Server_Gmd_Dir_Name	String	UNKNOWN
Server_Keep_Alive_Timeout	Numeric	15.0
Server_Max_Reconnect_Delay	Numeric	30.0
Server_Msg_Send	Boolean	TRUE
Server_Names	String List	_node
Server_Read_Timeout	Numeric	30.0
Server_Start_Delay	Numeric	1.0
Server_Start_Max_Tries	Numeric	1
Server_Start_Timeout	Numeric	30.0
Server_Write_Timeout	Numeric	30.0
Socket_Connect_Timeout	Numeric	5.0
Subjects	String List	None
Time_Format	Identifier	UNKNOWN
Trace_File	String	UNKNOWN
Trace_File_Size	Integer	0

Table 15 RTclient Options (Cont'd)

Option Name	Type	Default
Trace_Flags	String List	prefix
Trace_Level	String	UNKNOWN
Udp_Broadcast_Timeout	Numeric	5.0
Unique_Subject	String	_Node_Pid
Verbose	Boolean	FALSE

The options in this table that begin with Pgm are only valid for multicast. These options are documented in Chapter 10, Using Multicast.

RTserver Options Summary

The table summarizes the options available in RTserver. All of these options can be modified using the `setopt` command in the RTserver startup command file, or through a CONTROL message.

Table 16 RTserver Options

Option Name	Type	Default
Auth_Data_File	String	None
Authorize_Publish	Boolean	TRUE
Backup_Name	String	UNIX: ~ OpenVMS: None Windows: ~
Catalog_File	String	\$RTHOME/standard/tal_ss.cat
Catalog_Flags	String List	id
Client_Burst_Interval	Numeric	0.5
Client_Connect_Timeout	Numeric	10.0
Client_Drain_Subjects	Integer	1000
Client_Drain_Timeout	Real (seconds)	0.0
Client_Keep_Alive_Timeout	Numeric	0.0
Client_Max_Buffer	Numeric	10000000
Client_Max_Tokens	Numeric	0
Client_Read_Timeout	Numeric	0.0
Client_Reconnect_Timeout	Numeric	30.0
Client_Threads	Numeric	1
Client_Token_Rate	Numeric	0
Command_Feedback	Identifier	interactive

Table 16 RTserver Options (Cont'd)

Option Name	Type	Default
Compression	Boolean	FALSE
Compression_Args	String	6
Compression_Name	String	zlib
Compression_Stats	Boolean	FALSE
Conn_Max_Restarts	Numeric	0
Conn_Names	String List	UNIX: local, tcp OpenVMS: tcp Windows: tcp
Default_Connect_Prefix	Identifier	connect_one
Default_Msg_Priority	Numeric	0
Default_Protocols	Identifier List	UNIX: local, tcp OpenVMS: tcp Windows: tcp
Default_Subject_Prefix	String	/
Disable_Mon_Watch_Types	String List	None
Enable_Control_Msgs	String List	echo, quit
Enable_Stop_Msgs	Boolean	TRUE
Gmd_Publish_Timeout	Numeric	300.0
Group_Burst_Interval	Numeric	0.5
Group_Max_Buffer	Numeric	10000000
Group_Max_Tokens	Numeric	0
Group_Token_Rate	Numeric	0
Log_In_Client	String	None

Table 16 RTserver Options (Cont'd)

Option Name	Type	Default
Log_In_Group	String	None
Log_In_Server	String	None
Log_Out_Client	String	None
Log_Out_Group	String	None
Log_Out_Server	String	None
Max_Client_Conns	Numeric	200
Max_Server_Accept_Conns	Numeric	-1
Max_Server_Connect_Conns	Numeric	-1
Max_Server_Conns	Numeric	-1
Multi_Threaded_Mode	Boolean	FALSE
Proxy_Password	String	UNKNOWN
Proxy_Username	String	UNKNOWN
Real_Number_Format	String	%g
Server_Burst_Interval	Numeric	0.5
Server_Connect_Timeout	Numeric	10.0
Server_Connection_Names	String List	UNKNOWN
Server_Keep_Alive_Timeout	Numeric	15.0
Server_Max_Tokens	Numeric	0
Server_Names	String List	UNKNOWN
Server_Num_Threads	Numeric	1
Server_Read_Timeout	Numeric	30.0
Server_Reconnect_Interval	Numeric	30.0

Table 16 RTserver Options (Cont'd)

Option Name	Type	Default
Server_Threads	Numeric	0
Server_Token_Rate	Numeric	0
Sm_Security_Driver	String	None
Socket_Connect_Timeout	Numeric	5.0
Srv_Client_Names_Min_Msgs	Boolean	FALSE
Srv_Subj_Names_Min_Msgs	Boolean	FALSE
Time_Format	Identifier	hms
Trace_File	String	UNKNOWN
Trace_File_Size	Integer	0
Trace_Flags	String List	prefix
Trace_Level	String	UNKNOWN
Udp_Broadcast_Timeout	Numeric	5.0
Unique_Subject	String	_Node_Pid
Verbose	Boolean	FALSE
Zero_Recv_Gmd_Failure	Boolean	FALSE

RTmon Options Summary

The table summarizes the options available in RTmon. Because RTmon is a type of RTclient, most of the RTclient options are supported for RTmon and work in the same way for both RTclient and RTmon. All of the RTmon options can be modified using the `setopt` command from the RTmon command interface.

Table 17 RTmon Options

Option Name	Type	Default
Auth_Data_File	String	None
Backup_Name	String	UNIX: ~ OpenVMS: None Windows: ~
Catalog_File	String	\$RTHOME/standard/tal_ss.cat
Catalog_Flags	String List	id
Command_Feedback	Identifier	interactive
Compression	Boolean	FALSE
Compression_Args	String	6
Compression_Name	String	zlib
Compression_Stats	Boolean	FALSE
Default_Msg_Priority	Numeric	0
Default_Protocols	Identifier List	UNIX: local, tcp OpenVMS: tcp Windows: tcp
Default_Subject_Prefix	String	None
Editor	String	UNIX: vi OpenVMS: edt Windows: notepad

Table 17 RTmon Options

Option Name	Type	Default
Enable_Control_Msgs	String List	echo,quit
Ipc_Gmd_Directory	String	UNIX: /tmp/rtworks OpenVMS: sys\$scratch Windows: %TEMP%\rtworks
Ipc_Gmd_Type	String	default, which means file-based GMD is attempted
Log_In_Data	String	None
Log_In_Internal	String	None
Log_In_Msgs	String	None
Log_In_Status	String	None
Log_Out_Data	String	None
Log_Out_Internal	String	None
Log_Out_Msgs	String	None
Log_Out_Status	String	None
Monitor_Scope	String	/*
Project	Identifier	rtworks
Prompt	String	"Mon> "
Proxy_Password	String	UNKNOWN
Proxy_Username	String	UNKNOWN
Real_Number_Format	String	%g
Server_Auto_Connect	Boolean	TRUE
Server_Auto_Flush_Size	Numeric	8192
Server_Delivery_Timeout	Numeric	30.0

Table 17 RTmon Options

Option Name	Type	Default
Server_Disconnect_Mode	Identifier	gmd_failure
Server_Keep_Alive_Timeout	Numeric	15.0
Server_Max_Reconnect_Delay	Numeric	30.0
Server_Msg_Send	Boolean	TRUE
Server_Names	String List	_node
Server_Read_Timeout	Numeric	30.0
Server_Start_Delay	Numeric	1.0
Server_Start_Max_Tries	Numeric	1
Server_Start_Timeout	Numeric	30.0
Server_Write_Timeout	Numeric	30.0
Socket_Connect_Timeout	Numeric	5.0
Subjects	String List	None
Time_Format	Identifier	hms
Trace_File	String	UNKNOWN
Trace_File_Size	Integer	0
Trace_Flags	String List	prefix
Trace_Level	String	UNKNOWN
Udp_Broadcast_Timeout	Numeric	5.0
Unique_Subject	String	_Node_Pid
Verbose	Boolean	FALSE

Multi-Thread Mode

In single-thread mode, the server uses one thread for all client and server connections. In multi-thread mode, the server distributes connections to several I/O threads.

Multi-thread mode is enhanced in Release 6.7 (and later). This enhancement lets you configure two separate pools of threads:

- a client thread pool, for connections from clients with ordinary traffic levels
- a server thread pool, for connections from servers and high-traffic clients (such as TIBCO SmartSockets Cache, which handles volume comparable to a server)

Four options configure this feature:

- `Multi_Threaded_Mode`
- `Client_Threads`
- `Server_Threads`
- `Server_Connection_Names`

Special Values

If either `Client_Threads` or `Server_Threads` is non-zero, then RTserver operates in multi-thread mode.

If one (but not both) of `Client_Threads` or `Server_Threads` is zero, then *all* connections use I/O threads from the other (non-zero) pool.

The default configuration (zero for both `Client_Threads` and `Server_Threads`) specifies single-thread mode—one I/O thread services all client and server connections. (This default preserves backward compatibility.)



Although the server does not enforce a maximum number of threads in these pools, we advise caution when using large values; larger values do not necessarily result in better performance.

The optimal number of threads depends on the operational parameters of your deployment, such as the number of processors, the subject namespace, message fan-out characteristics, message rate, disk I/O rate, compression, the number of RTserver-to-RTclient connections, and the number of RTserver-to-RTserver connections (cloud configuration).

We recommend that you empirically determine the optimal size of the thread pools for your deployment. We suggest that you begin tuning with the values in Table 19, and adjust them based on the results of your testing. We caution that the optimal values can change significantly when the operational parameters of your deployment vary; when they do change, we strongly recommend that you re-test to determine the best values.

Table 18 Tuning the Number of I/O Threads

Number of CPUs	Initial Number of Total Threads
2–3	Our testing has not shown any benefit to adjusting these parameters on computers with fewer than 4 processors. We recommend single-thread mode.
4 or more	<p>We recommend that you begin tuning by setting Client_Threads and Server_Threads so that their sum is the number of CPUs plus 1. The way in which you allocate that total to clients and servers will depend on the needs of your deployment.</p> <p>For example, an RTserver routing hub running on an 8-CPU computer might allocate 2 threads to server connections, and 7 threads to client connections (for a total of 9 threads).</p>

To run in multi-thread mode, your RTserver must be licensed for the SmartSockets MP option. Furthermore, multi-thread mode is not available on all platforms. Attempting to configure multi-thread mode without an appropriate license or on a platform that does not support it, results in a warning message when RTserver starts, and it starts in single-thread mode.

This enhancement supersedes the deprecated option Server_Num_Threads.

Option Reference

Auth_Data_File

Used for:	RTclient, RTserver, RTmon, and RTgms processes
Type:	String
Default Value:	None
Valid Values:	Any valid filename

The Auth_Data_File option specifies the file containing file-based credentials. This option is useful for legacy applications that need to send credentials to a security driver. A credentials file may be created using RTacl.

Authorize_Publish

Used for:	RTserver
Type:	Boolean
Default Value:	TRUE
Valid Values:	TRUE or FALSE

When a security driver is installed, the Authorize_Publish option specifies whether the RTserver is required to authorize every RTclient publish. Setting this option to FALSE is equivalent to authorizing all publishes regardless of subject and message type. The security driver is not queried for authorization.

Backup_Name

Used for: RTclient, RTserver, RTmon, RTgms processes

Type: String

Default Value:

- UNIX and Windows: ~
- OpenVMS: none

Valid Values: Any valid filename characters

The Backup_Name option specifies the extension given to a backup file created when a file is opened in write mode. This includes all files created in the RT process (RTclient, RTserver, or RTmon) except for those created in RTsdb and view files.

The backup file has the same name as the existing file with the addition of the extension specified in this option. For example, if the default value of Backup_Name is used on a UNIX system, a file named `satellite1` would have a backup file named `satellite1~`. To turn off the creation of backup files, set the option to UNKNOWN.

If, while running RTclient, a file becomes corrupted, the backup file can be renamed (by dropping the extension) to recover the earlier version.

This option must precede any other file options (including Trace_File) within the command file.

Catalog_File

Used for: RTserver, RTclient, and RTmon processes

Type: String

Default Value: \$RTHOME/standard/tal_ss.cat

Valid Values: Any valid catalog filename

The Catalog_File option specifies the filename used for the SmartSockets resource catalog. In general, use the default name.

Catalog_Flags

Used for:	RTserver, RTclient, and RTmon processes
Type:	String List (Identifiers)
Default Value:	id
Valid Values:	id or unknown

The `Catalog_Flags` option specifies how to format the catalog strings. If you specify `id`, the catalog string identifier is included. Unsetting the option using either `unsetopt` or using `setopt` to change the value to `unknown` clears all the flags.

Client_Burst_Interval

Used for:	RTserver only
Type:	Real Number
Default Value:	0.5
Valid Values:	Any real number from 0.0 to 600.0, inclusive

The `Client_Burst_Interval` option specifies the burst interval in seconds used for a connection to an RTclient. When the RTserver has used up all its tokens on an RTclient connection (there are 0 tokens accumulated), the RTserver must wait for this burst interval before checking for more tokens to determine if it can send more data on its RTclient connection.

For more information, see the related options `Client_Max_Tokens` and `Client_Token_Rate`.

Client_Connect_Timeout

Used for:	RTserver and RTgms only
Type:	Real Number
Default Value:	10.0
Valid Values:	Any real number greater than 0

The `Client_Connect_Timeout` option specifies the maximum amount of time (in seconds) the RTserver or RTgms process waits when trying to read a `CONNECT_CALL` initialization message from a new RTclient process. If the RTserver or RTgms does not receive the message within the timeout period, it destroys the connection to the new RTclient process.

This option is required and cannot be unset.

Client_Drain_Subjects

Used for:	RTserver
Type:	Integer
Default Value:	1000
Valid Values:	Any integer greater than 0

When a client disconnects, the server must clean up its destroyed subscriptions. When the number of destroyed subjects is very large, the clean-up task could interfere with the server's responsiveness to active clients. Two options let you spread the clean-up effort over time, so active clients still receive prompt service:

- `Client_Drain_Timeout` specifies the interval to wait between clean-up sessions.
- `Client_Drain_Subjects` specifies the number of subjects to remove at each interval—until the server has removed all destroyed subjects.

Client_Drain_Timeout

Used for:	RTserver
Type:	Real Number
Default Value:	0.0
Valid Values:	Any real number in the range [0.0, 60.0]

When a client disconnects, the server must clean up its destroyed subscriptions. When the number of destroyed subjects is very large, the clean-up task could interfere with the server’s responsiveness to active clients. Two options let you spread the clean-up effort over time, so active clients still receive prompt service:

- Client_Drain_Timeout specifies the interval to wait between clean-up sessions.
- Client_Drain_Subjects specifies the number of subjects to remove at each interval—until the server has removed all destroyed subjects.

The Client_Drain_Timeout option specifies the interval (in seconds) to wait before unsubscribing subjects from destroyed RTclient connections.

When Client_Drain_Timeout is set to 0.0 (the default), this functionality is disabled—the server cleans up all the destroyed subjects in a single batch.

Client_Keep_Alive_Timeout

Used for:	RTserver only
Type:	Real Number
Default Value:	0.0
Valid Values:	Any real number 0.0 or greater

The Client_Keep_Alive_Timeout option specifies how long (in seconds) to wait when checking if an RTclient is still alive. If more than Client_Read_Timeout seconds have occurred since the RTserver last heard from the RTclient, the RTserver sends a keep alive message to the RTclient. The RTserver waits for the number of seconds you specify here in Client_Keep_Alive_Timeout to hear back from the RTclient. If the RTserver does not hear back from the RTclient in that time, the RTserver destroys the connection to that RTclient. When Client_Keep_Alive_Timeout is set to 0.0, the keep alives are disabled.

If you enable the server-to-client keep alives by setting Client_Keep_Alive_Timeout and Client_Read_Timeout to values other than 0.0, the RTclients connected to this RTserver must be able to process the keep alive messages from the RTserver, using either TipcSrvMsgProcess or TipcSrvMainLoop.

Client_Max_Buffer

Used for:	RTserver and RTgms only
Type:	Integer
Default Value:	10000000
Valid Values:	Any integer greater than 0

The Client_Max_Buffer option specifies the maximum number of bytes of data that are allowed to be buffered to each RTclient process. This maximum buffer size is used to check for possible network failures. To prevent a large backlog from consuming all available process memory, RTserver limits the buffer size of each RTclient connection to Client_Max_Buffer bytes of data; if a client or server connection consumes too slowly, so that the buffer grows beyond this limit, then RTserver destroys the connection.

This option is required and cannot be unset.

Client_Max_Tokens

Used for:	RTserver only
Type:	Integer
Default Value:	0
Valid Values:	Any integer from 0 to 2147483647, inclusive

The Client_Max_Tokens option specifies the maximum number of tokens that can accumulate for connections to RTclients. The default value of 0 specifies that an unlimited number of tokens can accumulate.

For more information, see the related options Client_Burst_Interval and Client_Token_Rate.

Client_Read_Timeout

Used for:	RTserver only
Type:	Real Number
Default Value:	0.0
Valid Values:	Any real number 0.0 or greater

The Client_Read_Timeout option specifies how often (in seconds) data is expected to be available for reading on a connection between an RTserver and an RTclient. This timeout is used to check for possible network failures or ghost clients. If a read timeout occurs, the RTserver sends a keep alive message to the RTclient. The RTserver waits for the number of seconds you specify in Client_Keep_Alive_Timeout to hear back from the RTclient. If the RTserver does not hear back from the RTclient in that time, the RTserver destroys the connection to that RTclient. Checking for read timeouts is disabled if Client_Read_Timeout is set to 0.0. The larger the value for Client_Read_Timeout, the longer the RTserver waits to detect a possible RTclient failure.

If you enable the server-to-client keep alives by setting Client_Keep_Alive_Timeout and Client_Read_Timeout to values other than 0.0, the RTclients connected to this RTserver must be able to process the keep alive messages from the RTserver, using either TipcSrvMsgProcess or TipcSrvMainLoop.

Helpful Read Timeout Tips

If you are enabling keep alives with a read timeout, you can reduce network traffic by setting the options `Client_Read_Timeout` and `Server_Read_Timeout` to different values. If they are set to the same value, other than `0.0`, the `RTclient` and `RTserver` send a keep-alive message at the same time to each other. This is unnecessary, as only one keep-alive message and response is needed to keep the connection open.

The RT process you set to have the smaller read timeout, whether it is the `RTclient` or the `RTserver`, becomes the process that generally sends the keep-alive messages. For more efficient `RTserver` processing, we recommend that you set the value for `Server_Read_Timeout` to be smaller than the value for `Client_Read_Timeout`, or set `Client_Read_Timeout` to `0.0` to disable server-to-client keep alives. This places the burden of sending keep-alives on the `RTclient`, which only has one `RTserver` to check on, instead of on the `RTserver`, which might support many `RTclients`.

Client_Reconnect_Timeout

Used for:	RTserver only
Type:	Real Number
Default Value:	30.0
Valid Values:	Any real number greater than 0

The `Client_Reconnect_Timeout` option specifies the maximum amount of time (in seconds) `RTserver` waits for a warm `RTclient` to reconnect after the `RTclient` disconnects for any reason from `RTserver`. The option `Server_Disconnect_Mode` in `RTclient` controls whether or not all `RTserver` processes perform this wait. If `Server_Disconnect_Mode` is `warm`, all `RTserver` processes save the subject information about the `RTclient` and buffer messages for GMD so no messages are lost. If `Server_Disconnect_Mode` is `gmd_failure` or `gmd_success`, then no waiting takes place.

RTserver does not synchronously wait for the RTclient to reconnect, but instead includes the timeout in its main processing loop. If the warm RTclient does not reconnect to RTserver within Client_Reconnect_Timeout seconds, then RTserver clears the guaranteed messages that have not been acknowledged by the RTclient process and sends a GMD_NACK message back to the sender of these messages. The warm RTclient can reconnect to this RTserver or any other RTserver that this RTserver is connected to (either directly or indirectly).

This option is required and cannot be unset.

Client_Threads

Used for:	RTserver process only
Type:	Integer
Default Value:	0
Valid Values:	Any non-negative integer

Client_Threads determines the number of threads in the pool for client connections.



For important background information and the semantics of special values, see [Multi-Thread Mode](#) on page 512.

This option is one of four that replace the deprecated option `Server_Num_Threads`:

- `Multi_Threaded_Mode`
- `Server_Threads`
- `Client_Threads`
- `Server_Connection_Names`

Client_Token_Rate

Used for:	RTserver only
Type:	Integer
Default Value:	0
Valid Values:	Any integer from 0 to 2147483647, inclusive

The `Client_Token_Rate` option specifies the rate, in bytes a second, at which tokens accumulate for `RTclient` connections. Specifying a value of 0 disables client bandwidth control.

For more information, see the related options `Client_Burst_Interval` and `Client_Max_Tokens`.

Command_Feedback

Used for:	RTclient, RTserver, RTmon processes
Type:	String (Identifier)
Default Value:	interactive
Valid Values:	always OR interactive OR never

The `Command_Feedback` option specifies when feedback is displayed after a command is successfully executed. If the command results in an error such as incorrect syntax, feedback is always displayed, regardless of the value of `Command_Feedback`. The three possible values for `Command_Feedback` are:

<code>always</code>	feedback is always given, regardless of how the command was executed, interactively or using a <code>CONTROL</code> message.
<code>interactive</code>	feedback is only given when the command is entered interactively.
<code>never</code>	feedback is never given.

Setting the option to `always` is useful for learning more about how the RT process (`RTclient`, `RTserver`, or `RTmon`) works and if any commands are being sent in from other processes with `CONTROL` messages. A value of `never` is the most efficient.

This option is required and cannot be unset.

Compression

Used for:	RTclient, RTserver, RTmon, RTgms processes
Type:	Boolean
Default Value:	FALSE
Valid Values:	TRUE or FALSE

The Compression option specifies whether connection-level compression is enabled. If set to TRUE then all data sent on all connections is compressed. An alternative to using the Compression option is to set the compression property of the enhanced LCN in either the Conn_Names option or the Server_Names option. The actual compression algorithm used is specified by the Compression_Name option.

Compression_Args

Used for:	RTclient, RTserver, RTmon, RTgms processes
Type:	String
Default Value:	6
Valid Values:	Valid arguments for the library specified by Compression_Name

The Compression_Args option allows arguments specific to the compression library specified by the Compression_Name option to be passed to it. Currently, the only available compression library is ZLIB. The valid argument for the ZLIB library is an integer value from 1 to 9, which specifies the compression level. 1 gives the best speed, 9 gives the best compression.

Compression_Name

Used for:	RTclient, RTserver, RTmon, RTgms processes
Type:	String
Default Value:	zlib
Valid Values:	Any valid SmartSockets compression library

The Compression_Name option specifies what SmartSockets compression library is used to perform connection-level compression and message compression. Currently, the only available compression library is ZLIB.

Compression_Stats

Used for:	RTclient, RTserver, RTmon, RTgms processes
Type:	Boolean
Default Value:	FALSE
Valid Values:	TRUE or FALSE

The Compression_Stats option specifies whether compression statistics are printed. When set to TRUE, compression statistics are printed approximately every 30 seconds.



Enabling compression statistics is intended for debugging purposes only because it lowers performance.

Conn_Max_Restarts

Used for:	RTserver only
Type:	Integer
Default Value:	0
Valid Values:	Any integer 0 or greater

The Conn_Max_Restarts option specifies the maximum number of times that RTserver restarts a server connection if an error occurs while accepting a new RTclient or RTserver. If RTserver does not restart the connection, then the connection cannot be used to rendezvous with any new processes. These errors are rare but can occur if a network or system failure occurs. If Conn_Max_Restarts is set to 0, then RTserver always restarts a server connection. For a discussion of how a server connection accepts a client connection, see The Server Accepts the Client on page 105.

This option is required and cannot be unset.

Conn_Names

Used for:	RTserver and RTgms only
Type:	String List
Default Value:	For RTserver: <ul style="list-style-type: none">• UNIX: local, tcp• Windows: tcp• OpenVMS: tcp For RTgms: <ul style="list-style-type: none">• UNIX: pgm:_node:local.5104, pgm:_node:tcp.5104• Windows: pgm:_node:tcp.5104
Valid Values:	Any valid logical connection name or names, separated by commas (,)

The Conn_Names option specifies a list of logical connection names used by RTclient processes and other RTserver processes to find this RTserver or used by RTclient processes to find this RTgms process. Each logical connection name has either of these forms:

- *protocol:node:address*, which can be shortened to *protocol:node*, *protocol*, or *node*, for most normal connections to an RTserver

When *protocol* is `tcp` or a TCP-based protocol on a multi-homed machine, you can set *node* to either of these keywords:

`_node` causes RTserver to listen only on the default IP address for the machine.

`_any` causes RTserver to listen on all IP addresses for the machine.

- *pgm:node:unicast_protocol.address*, used when the logical connection is for multicast, to enable RTclients to find this RTgms



For Conn_Names, you must specify the full domain name for *node* when using `pgm`.

- *proxy:node.address@protocol:dest_node:dest_address*, which is used when the logical connection is through a proxy server (for example, a web proxy server with HTTP CONNECT enabled) to the RTserver.

The logical connection names are separated by commas (.). For more information about valid protocols, nodes, and addresses, see Logical Connection Names for RT Processes on page 192.

This option is required and cannot be unset.



Using the keyword `_any` in Conn_Names is discouraged for RTserver to RTserver connections. When an RTserver connects to another RTserver whose Conn_Names use `_any`, the RTserver might attempt to reconnect every `Server_Reconnect_Interval` seconds. This is a known problem and will be fixed in a future release.

Default_Connect_Prefix

Used for:	RTserver only
Type:	String (Identifier)
Default Value:	connect_one
Valid Values:	<ul style="list-style-type: none">connect_allconnect_oneconnect_all_stopconnect_one_stop

The `Default_Connect_Prefix` option specifies the default connect prefix to use when one is not specified in a value in the `Server_Names` option. There are four possible values:

<code>connect_all</code>	connect to all RTservers that the new RTserver is connected to.
<code>connect_one</code>	connect only to the new RTserver, and not to all the RTservers it is connected to.
<code>connect_all_stop</code>	connect to all RTservers that the new RTserver is connected to, and stop traversing <code>Server_Names</code> if the first connection succeeded (most closely emulates Version 3.5 behavior).
<code>connect_one_stop</code>	connect only to the new RTserver, not to all the RTservers it is connected to, and stop traversing <code>Server_Names</code> if the first connection succeeded.

Using a connect prefix of `connect_all` causes all RTservers to interconnect as much as possible, while a connect prefix of `connect_one` allows precise control over the RTserver topology.

Default_Msg_Priority

Used for:	RTclient, RTserver, RTmon, and RTgms processes
Type:	Integer
Default Value:	0
Valid Values:	Any integer between -32768 and 32767, inclusive

The Default_Msg_Priority option specifies the default priority for newly created messages. The message priority property controls where an incoming message is inserted into a connection's message queue. When a message is created, its priority is initialized to the message type priority (if set) or to Default_Msg_Priority (if the message type priority is unknown).

Default_Protocols

Used for:	RTclient, RTserver, RTmon processes
Type:	String (Identifier) List
Default Value:	<ul style="list-style-type: none"> • UNIX: local, tcp • Windows: tcp • OpenVMS: tcp
Valid Values:	local, tcp

The Default_Protocols option specifies a list of IPC protocols to try if no protocol is listed in a logical connection name in the Server_Names or Conn_Names options. For further information regarding protocols, see [Creating a Connection to RTserver](#) on page 189.

Default_Subject_Prefix

Used for:	RTclient, RTserver, RTmon, and RTgms processes
Type:	String
Default Value:	<ul style="list-style-type: none">RTserver: /RTclient, RTmon, RTgms: UNKNOWN
Valid Values:	Any valid prefix character, and also UNKNOWN for RTclient, RTmon, or RTgms

The Default_Subject_Prefix option specifies the qualifier to prefix to message subject names that do not start with /. Subject names are organized in a hierarchical namespace where the components are delimited by /. A subject name that starts with / is called an absolute subject name. All non-absolute subject names have Default_Subject_Prefix prefixed to them so as to create a fully qualified name for the hierarchical subject namespace. For more information about Default_Subject_Prefix, see Subjects on page 158.

A default prefix is required for all RTservers, whether you use the default value for RTserver (/) or set it to another value. When specifying this value for an RTserver, do not unset this value or set it to UNKNOWN.

The other RTprocesses, RTclient, RTmon and RTgms, use the value set for the first RTserver they connect to if their own value for Default_Subject_Prefix is unset or set to UNKNOWN.

Disable_Mon_Watch_Types

Used for:	RTserver
Type:	String List
Default Value:	None
Valid Values:	<ul style="list-style-type: none"> • One or more of the <code>watch</code> values described in <code>watch</code> on page 634, separated by commas (,) • <code>_all</code> • <code>UNKNOWN</code>

The `Disable_Mon_Watch_Types` option specifies a comma-separated list of watch types to be ignored by the server. Legal values are as accepted by the `RTmon watch` command listed in `watch` on page 634 or `_all` for all watches.



If an item in the list is prefixed with '!', then the behavior is negated. For example, the value `!client_names,_all` will disable all watches except `client_names`.

There is no client notification if a watch is denied by the server (the client will think it is watching, but will receive no messages), however a warning message is emitted to the trace log.



The value `UNKNOWN` is a keyword. Do not enclose it in double quotation characters (").

A warning is emitted upon rejecting a watch.

Editor

Used for:	RTclient, RTmon, and RTgms processes
Type:	String
Default Value:	<ul style="list-style-type: none">• UNIX: <code>vi</code>• Windows: <code>notepad</code>• OpenVMS: <code>edt</code>
Valid Values:	Any valid program or shell script name

The Editor option specifies which text editor (or program) is used when the RT process initiates an editor session. Depending on the system configuration, possible values for this option include `emacs`, `vi` (for UNIX), `edt` (for OpenVMS), `Notepad`, `write` (for Windows), and other editors such as `textedit` (for Solaris).

Also, any program or shell script can be specified as the value of the Editor option. For example, if you specified `cat` as the value:

```
CLIENT> setopt editor cat
```

the name of the file is fed to the `cat` program and the file is output to standard output. This option is required and cannot be unset.

Enable_Control_Msgs

Used for:	RTclient, RTserver, RTmon, and RTgms processes
Type:	String List
Default Value:	echo, quit
Valid Values:	<ul style="list-style-type: none"> Any valid command name or names, separated by commas (.) _all UNKNOWN

The `Enable_Control_Msgs` option is a string list specifying the commands allowed in a `CONTROL` message. The default allows the inclusion of the `echo` and `quit` commands in a `CONTROL` message. When this option is set to `UNKNOWN`, all commands (including `echo` and `quit`) are disabled. Setting this option to `_all` allows the inclusion of all valid commands in a `CONTROL` message.



If an item in the list is prefixed with '!', then the behavior is negated. For example, the value `!setopt,_all` would allow all control messages except `setopt`.

Security-conscious sites should set this option carefully, as inadvertent or malicious misuse can cause damage to the system, the data, or the application. There are several commands that may be dangerous when executed. Specifically, the `alias`, `connect`, `disconnect`, `retrieve`, `setopt`, `sh`, `source`, `subscribe`, `unalias`, `unsetopt`, and `unsubscribe` commands should only be specified with extreme caution.



The value `UNKNOWN` is a keyword. Do not enclose it in double quotations characters (")

A warning is emitted upon rejecting a control message.

Enable_Stop_Msgs

Used for:	RTserver only
Type:	Boolean
Default Value:	TRUE
Valid Values:	TRUE or FALSE

The `Enable_Stop_Msgs` option specifies whether or not RTserver can be stopped with `rtserver -stop`. By default this option is enabled, allowing you and all users to stop RTserver.

When disabled, no variation of the `rtserver -stop` command is permitted (including `rtserver -stop_all`).

Security-conscious sites should set this option to `FALSE` to prevent accidental shutdown of an entire SmartSockets system.

This option is required and cannot be unset.

Gmd_Publish_Timeout

Used for:	RTserver only
Type:	Real Number
Default Value:	300.0
Valid Values:	Any real number 0.0 or greater

The `Gmd_Publish_Timeout` option specifies the amount of time RTserver continues to maintain GMD information for a subject that has not been recently published to with GMD. There is some initial overhead for the first publish to a subject using GMD and then a smaller amount of overhead to maintain the GMD accounting information. If no direct RTclients publish to a subject with GMD in `Gmd_Publish_Timeout` seconds, and if all direct publishing RTclients disconnect, then RTserver stops maintaining this GMD accounting until the next GMD publish occurs. Checking for GMD publishing timeouts is disabled if `Gmd_Publish_Timeout` is set to `0.0`.

This option is required and cannot be unset.

Group_Burst_Interval

Used for:	RTserver and RTgms processes only
Type:	Real Number
Default Value:	0.5
Valid Values:	Any real number from 0.0 to 600.0, inclusive

The Group_Burst_Interval option specifies the burst interval in seconds used for an outbound group channel connection.

If you set this option for RTserver, when the RTserver has used up all its tokens on a group channel connection (there are 0 tokens accumulated), the RTserver must wait for this burst interval before checking for more tokens to determine if it can send more data to RTgms.

If you set this option for RTgms, when the RTgms has used up all its tokens on a group channel connection (there are 0 tokens accumulated), the RTserver must wait for this burst interval before checking for more tokens to determine if it can send more data to the RTserver.

To control bandwidth in both directions, you must set this option for both RTgms and for RTserver.

For more information, see the related options Group_Max_Tokens and Group_Token_Rate.

Group_Max_Buffer

Used for:	RTserver and RTgms only
Type:	Integer
Default Value:	10000000
Valid Values:	Any integer greater than 0

The `Group_Max_Buffer` option specifies the maximum number of bytes of data that are allowed to be buffered to each group connection (such as the one used by RTgms—the multicast publishing process). This maximum buffer size is used to check for possible network failures. To prevent a large backlog from consuming all available process memory, RTserver limits the buffer size of each group connection to `Group_Max_Buffer` bytes of data; if a group connection (such as a connection from RTgms) consumes too slowly, so that the buffer grows beyond this limit, then RTserver destroys the connection.

This option is required and cannot be unset.

Group_Max_Tokens

Used for:	RTserver and RTgms processes only
Type:	Integer
Default Value:	0
Valid Values:	Any integer from 0 to 2147483647, inclusive

The `Group_Max_Tokens` option specifies the maximum number of tokens that can accumulate for group channel connections. The default value of 0 specifies that an unlimited number of tokens can accumulate.

If you specify this option for RTserver, it sets the maximum tokens for RTserver to use to send messages along group channels to RTgms processes. If you specify this option for RTgms, it sets the maximum tokens for RTgms to use to send multicast messages along group channels to the RTserver. To control bandwidth in both directions, you must set this option for both RTgms and for RTserver.

For more information, see the related options `Group_Burst_Interval` and `Group_Token_Rate`.

Group_Names

Used for:	RTclient and RTgms processes only
Type:	String List
Default Value:	rtworks
Valid Values:	Any valid multicast group names or multicast addresses, separated by commas

The Group_Names option specifies a list of multicast groups or multicast addresses. This is the list of groups to which this RTclient belongs, and indicates to the RTgms for that RTclient how to route multicast messages for this RTclient.

If your RTclient belongs to several multicast groups, you can specify a mix of multicast group names and addresses as needed:

```
setopt group_names fred,224.10.10.10,stockg
```

Specify the multicast group name as an address only if you do not want the value to be hashed.

This option is optional, and is only used if you have a license for the SmartSockets Multicast.

Group-Token_Rate

Used for:	RTserver and RTgms processes only
Type:	Integer
Default Value:	0
Valid Values:	Any integer from 0 to 2147483647, inclusive

The Group-Token_Rate option specifies the rate, in bytes a second, at which tokens accumulate for outbound group channel connections. Specifying a value of 0 disables group bandwidth control.

If you specify this option for RTserver, it controls the tokens accumulating for RTserver to send messages along group channels to RTgms processes. If you specify this option for RTgms, it controls the tokens accumulating for RTgms to send multicast messages along group channels to the RTserver. To control bandwidth in both directions, you must set this option for both RTgms and for RTserver.

For more information, see the related options Group_Burst_Interval and Group_Max_Tokens.

Ipc_Gmd_Auto_Ack

Used for:	RTclient
Type:	Boolean
Default Value:	TRUE
Valid Values:	TRUE or FALSE

The Ipc_Gmd_Auto_Ack option enables and disables automatic acknowledgment of GMD messages. If set to TRUE, all received GMD messages are automatically acknowledged when the message is destroyed. If disabled set to FALSE, the user must explicitly acknowledge receipt of the GMD message by calling TipcMsgAck.

Use the Ipc_Gmd_Auto_Ack_Policy option to specify when a message is automatically acknowledged.

This option is required and cannot be unset.

ipc_Gmd_Auto_Ack_Policy

Used for:	RTclient
Type:	String
Default Value:	first_destroy
Valid Values:	first_destroy or last_destroy

The `ipc_Gmd_Auto_Ack_Policy` option specifies when a GMD message is automatically acknowledged. There are two possible values:

`first_destroy` the received GMD message is acknowledged the first time the message is destroyed regardless of its reference count.

`last_destroy` the received GMD message is acknowledged when its reference count falls to zero.

To enable or disable automatic acknowledgement of GMD messages, use the `ipc_Gmd_Auto_Ack` option.

This option is required and cannot be unset.

ipc_Gmd_Directory

Used for:	RTclient, RTmon, and RTgms processes
Type:	String
Default Value:	<ul style="list-style-type: none"> UNIX: <code>/tmp/rtworks</code> Windows: <code>%TEMP%\rtworks</code> OpenVMS: <code>sys\$scratch</code>
Valid Values:	Any valid directory name

The `ipc_Gmd_Directory` option specifies the directory used to store messages for file-based GMD. These messages are saved on disk so they can be resent after a delivery failure. File-based GMD is slower than memory-based GMD, because of the time needed to write to disk. For best performance, specify a local file system for your GMD directory. Read and write access to this directory is required.

This option is required and cannot be unset.

Ipc_Gmd_Type

Used for:	RTclient, RTmon, and RTgms processes
Type:	String
Default Value:	default
Valid Values:	default, memory or file

The `Ipc_Gmd_Type` specifies whether file-based or memory-based GMD is to be used. If left to its default value of `default`, file-based GMD is attempted, and if that is unsuccessful, memory-based GMD is used.

If the value is set to `memory`, memory-based GMD is used. If unable to set memory-based GMD, the connection fails. This option is only for use with GMD.

If the value is set to `file`, file-based GMD is used upon opening a connection. If unable to set file-based GMD, the connection fails. This option is only for use with GMD.

Log_In_Client

Used for:	RTserver and RTgms only
Type:	String
Default Value:	None
Valid Values:	Any valid file name

The `Log_In_Client` option specifies the name of the file that RTserver or RTgms uses to log incoming messages of all types that are received from RTclient processes. If this option is not set, incoming messages from RTclient processes are not logged.

Log_In_Data

Used for:	RTclient, RTmon, and RTgms processes
Type:	String
Default Value:	None
Valid Values:	Any valid file name

The Log_In_Data option specifies the name of the file that the RT process (RTclient, RTmon, or RTgms) uses to log incoming data messages, such as TIME or NUMERIC_DATA, that are received from RTserver. If this option is not set, incoming data messages are not logged. Data messages are listed in Chapter 3, Publish-Subscribe.

Log_In_Group

Used for:	RTserver and RTgms only
Type:	String
Default Value:	None
Valid Values:	Any valid file name

The Log_In_Group option specifies the name of the file that RTserver or RTgms uses to log all incoming messages that are received through multicast groups. If this option is not set, incoming messages from multicast groups are not logged.

Log_In_Internal

Used for:	RTclient, RTmon, and RTgms processes
Type:	String
Default Value:	None
Valid Values:	Any valid file name

The Log_In_Internal option specifies the name of a file that the RT process uses to log incoming internal messages, such as CONNECT_CALL or MON_SUBJECT_SUBSCRIBE_SET_WATCH, that are received from RTserver. If this option is not set, incoming internal messages are not logged. Internal messages are listed in Chapter 3, Publish-Subscribe.

Log_In_Msgs

Used for:	RTclient and RTmon processes
Type:	String
Default Value:	None
Valid Values:	Any valid file name

The Log_In_Msgs option specifies the name of a file that the RTclient uses to log all incoming messages. If this option is not set, incoming messages are not logged.

Log_In_Server

Used for: RTserver only
Type: String
Default Value: None
Valid Values: Any valid file name

The Log_In_Server option specifies the name of the file that RTserver uses to log incoming messages of all types that are received from other RTserver processes. If this option is not set, incoming messages from other RTserver processes are not logged.

Log_In_Status

Used for: RTclient, RTmon, and RTgms processes
Type: String
Default Value: None
Valid Values: Any valid file name

The Log_In_Status option specifies the name of a file that the RT process uses to log incoming status messages, such as ALERT or INFO, that are received from RTserver. If this option is not set, incoming status messages are not logged. Status messages are listed in Chapter 3, Publish-Subscribe.

Log_Out_Client

Used for:	RTserver and RTgms only
Type:	String
Default Value:	None
Valid Values:	Any valid file name

The Log_Out_Client option specifies the name of the file that RTserver or RTgms uses to log outgoing messages of all types that are sent to RTclient processes. If this option is not set, outgoing messages to RTclient processes are not logged.

Log_Out_Data

Used for:	RTclient, RTmon, and RTgms processes
Type:	String
Default Value:	None
Valid Values:	Any valid file name

The Log_Out_Data option specifies the name of a file that the RT process uses to log outgoing data messages, such as TIME or NUMERIC_DATA, that are sent to RTserver. If this option is not set, outgoing data messages are not logged. Data messages are listed in Chapter 3, Publish-Subscribe.

Log_Out_Group

Used for:	RTserver and RTgms only
Type:	String
Default Value:	None
Valid Values:	Any valid file name

The Log_Out_Group option specifies the name of the file that RTserver or RTgms uses to log outgoing messages of all types that are sent to multicast groups. If this option is not set, outgoing messages to multicast groups are not logged.

Log_Out_Internal

Used for:	RTclient, RTmon, and RTgms processes
Type:	String
Default Value:	None
Valid Values:	Any valid file name

The Log_Out_Internal option specifies the name of a file that the RT process uses to log outgoing internal messages, such as CONNECT_CALL or MON_SUBJECT_SUBSCRIBE_SET_WATCH, that are sent to RTserver. If this option is not set, outgoing internal messages are not logged. Internal messages are listed in Chapter 3, Publish-Subscribe.

Log_Out_Msgs

Used for:	RTclient and RTmon processes
Type:	String
Default Value:	None
Valid Values:	Any valid file name

The Log_Out_Msgs option specifies the name of a file that the RTclient uses to log all outgoing messages. If this option is not set, outgoing messages are not logged.

Log_Out_Server

Used for:	RTserver only
Type:	String
Default Value:	None
Valid Values:	Any valid file name

The Log_Out_Server option specifies the name of the file that RTserver uses to log outgoing messages of all types that are sent to other RTserver processes. If this option is not set, outgoing messages to other RTserver processes are not logged.

Log_Out_Status

Used for:	RTclient, RTmon, and RTgms processes
Type:	String
Default Value:	None
Valid Values:	Any valid file name

The Log_Out_Status option specifies the name of a file that the RT process uses to log outgoing status messages, such as ALERT or INFO, that are sent to RTserver. If this option is not set, outgoing status messages are not logged. Status messages are listed in Chapter 3, Publish-Subscribe.

Max_Client_Conns

Used for:	RTserver
Type:	Integer
Default Value:	200
Valid Values:	Any integer 0 or greater

The Max_Client_Conns option specifies the maximum number of RTclient processes that are allowed to connect to this RTserver process. If this limit is exceeded, no more RTclients are allowed to connect. While the techniques described in File Descriptor Upper Limit can be used to implement operating system-level brute force limit checking, Max_Client_Conns provides a more graceful way to enforce a maximum load on RTserver. Checking for the number of RTclients is disabled if Max_Client_Conns is set to 0.

This option is required and cannot be unset.

Max_Server_Accept_Conns

Used for:	RTserver only
Type:	Integer
Default Value:	-1
Valid Values:	Any integer

The `Max_Server_Accept_Conns` option specifies the maximum number of RTserver processes that are allowed to connect to this RTserver process. If this limit is exceeded, no more RTservers are allowed to connect.

`Max_Server_Accept_Conns` provides a way to limit connections made to an RTserver. Specifying a negative value allows an unlimited number of connections to be accepted by an RTserver. The default is an unlimited number of connections.

Related options are `Max_Server_Connect_Conns` and `Max_Server_Conns`. The value specified for `Max_Server_Conns` takes precedence over the value you specify for `Max_Server_Accept_Conns`. If you specify a larger value for `Max_Server_Accept_Conns` than you specify for `Max_Server_Conns`, the connections are limited by the value set for `Max_Server_Conns`.

This option is required and cannot be unset.

Max_Server_Connect_Conns

Used for:	RTserver only
Type:	Integer
Default Value:	-1
Valid Values:	Any integer

The `Max_Server_Connect_Conns` option specifies the maximum number of connections this RTserver can make to other RTserver processes. If this limit is exceeded, no more connections to other RTservers are allowed.

`Max_Server_Connect_Conns` provides a way to limit the number of server-to-server connections an RTserver requests. Specifying a negative value allows an unlimited number of connections to be initiated by an RTserver to other RTservers. The default is an unlimited number of connections.

Related options are `Max_Server_Accept_Conns` and `Max_Server_Conns`. The value specified for `Max_Server_Conns` takes precedence over the value you specify for `Max_Server_Connect_Conns`. If you specify a larger value for `Max_Server_Connect_Conns` than you specify for `Max_Server_Conns`, the connections are limited by the value set for `Max_Server_Conns`.

This option is required and cannot be unset.

Max_Server_Conns

Used for:	RTserver only
Type:	Integer
Default Value:	-1
Valid Values:	Any integer

The `Max_Server_Conns` option specifies the maximum number of server-to-server connections allowed for this RTserver. This limit includes connections initiated by other RTservers and connections to other RTservers initiated by this RTserver. If this limit is exceeded, no more server connections are allowed. `Max_Server_Conns` provides a way to enforce a maximum connection load on an RTserver. Specifying a negative value allows an unlimited number of server-to-server connections. The default is an unlimited number of connections.

Related options are `Max_Server_Accept_Conns` and `Max_Server_Connect_Conns`. The value specified for `Max_Server_Conns` takes precedence over the values specified for `Max_Server_Accept_Conns` and `Max_Server_Connect_Conns`. For example, if you specify 20 for your `Max_Server_Conns`, and 40 connections for `Max_Server_Connect_Conns`, the 21st connection, regardless of whether it is inbound or outbound, is not allowed.

This option is required and cannot be unset.

Monitor_Ident

Used for:	RTclient processes
Type:	String
Default Value:	RTclient
Valid Values:	Any valid string

The Monitor_Ident option sets the monitoring identification string for this process. The identification string is used as a descriptive name for the process when it is being monitored. Identification strings can also be set with the TipcMonSetIdentStr API call.

Monitoring is described in more detail in Chapter 5, Project Monitoring, on page 359. For more information about TipcMonSetIdentStr, see the *TIBCO SmartSockets Application Programming Interface*.

This string is stored in RTserver and is only sent to RTserver when RTclient connects to RTserver. A process that sets this option after connecting to RTserver will be identified incorrectly.

This option is required and cannot be unset.

Monitor_Level

Used for:	RTclient, RTmon, and RTgms processes
Type:	String
Default Value:	standard
Valid Values:	<ul style="list-style-type: none"> • none -- not implemented in this release • standard -- all monitoring except extended traffic monitoring • all -- all monitoring, including traffic monitoring and memory or cpu intensive monitoring

The Monitor_Level option sets the level of monitoring information that is maintained for this process. The monitoring level controls whether or not certain types of monitoring information that may be CPU or memory intensive are collected. This option must be set before a connection is created in order to have an effect.

Monitoring is described in more detail in Chapter 5, Project Monitoring, on page 359.

This option is required and cannot be unset.

Monitor_Scope

Used for:	RTclient, RTmon, and RTgms processes
Type:	String
Default Value:	/*
Valid Values:	Any valid subject name character or characters

The Monitor_Scope option specifies the level of interest for SmartSockets monitoring in those monitoring categories with no parameters, such as RTclient names poll or a parameter of @, such as RTclient time watch.

Monitor_Scope acts as a filter that can be used to prevent a large project from overloading a monitoring program. The default is /*, which matches all subject names at the first level of the hierarchical subject namespace. When Monitor_Scope is set to /. . ., which matches all names, all monitoring information is enabled, so all filtering is disabled. Monitoring scope is described in more detail in Chapter 5, Project Monitoring.

This option is required and cannot be unset.

Multi_Threaded_Mode

Used for: RTserver process only

Type: Boolean

Default Value: FALSE

Multi_Threaded_Mode enables (TRUE) or disables (FALSE) the multiple server threads. Disabling this feature results in one thread for all client and server connections.

If this option is FALSE, the server ignores the other three related options.

If Client_Threads or Server_Threads are set, but this option is not set, then the server outputs a warning.

In release 6.7, the deprecated option Server_Num_Threads overrides this option (in later releases, Server_Num_Threads will become obsolete, and will no longer override Multi_Threaded_Mode). Nonetheless, if Client_Threads or Server_Threads is set, they override Server_Num_Threads.



For important background information, see Multi-Thread Mode on page 512.

This option is one of four that replace the deprecated option Server_Num_Threads:

- Multi_Threaded_Mode
- Server_Threads
- Client_Threads
- Server_Connection_Names

Project

Used for:	RTclient and RTmon processes
Type:	String (Identifier)
Default Value:	rtworks
Valid Values:	Any valid project name

The Project option specifies the name of the SmartSockets project to which the RT process (RTclient or RTmon) is connected. The RT process (RTclient or RTmon) can only communicate with other SmartSockets processes that have the same project name.

This option is required and cannot be unset.

Prompt

Used for:	RTmon processes
Type:	String
Default Value:	RTmon: "MON> "
Valid Values:	Any valid character string

The Prompt option specifies the string that the RTmon process uses to prompt users for commands. It is best to keep the string short, under ten characters, so that your prompt does not take up too much of your screen's line length.

Proxy_Password

Used for:	RTserver, RTclient, and RTmon processes
Type:	String
Default Value:	UNKNOWN
Valid Values:	Any valid user password

The Proxy_Password option is a string that specifies the user password that the RT process (RTclient, RTserver, or RTmon) provides to a proxy server for authentication. This option is only needed if your RT process is going to attempt to connect to a proxy server.

When connecting to a proxy server that requires authentication, both Proxy_Password and Proxy_Username must be set. If the proxy server does not require authentication, then neither option is needed.

Proxy_Username

Used for:	RTserver, RTclient, and RTmon processes
Type:	String
Default Value:	UNKNOWN
Valid Values:	Any valid username

The Proxy_Username option is a string that specifies the username that the RT process (RTclient, RTserver, or RTmon) provides to a proxy server for authentication. This option is only needed if your RT process is going to attempt to connect to a proxy server.

When connecting to a proxy server that requires authentication, both Proxy_Password and Proxy_Username must be set. If the proxy server does not require authentication, then neither option is needed.

Real_Number_Format

Used for:	RTserver, RTclient, RTmon, and RTgms processes
Type:	String
Default Value:	%g
Valid Values:	Any valid <code>printf</code> conversion string

The `Real_Number_Format` option is a string that specifies the format that the RT process uses to print real numbers. The format string should be a `printf` (a standard C function) conversion string.

This option is required and cannot be unset.

Sd_Basic_Acl

Used for:	<code>sdbasic.cm</code>
Type:	String
Default Value:	<code>\$RTHOME/acl</code>
Valid Values:	Any valid directory containing the basic ACL configuration files

The `Sd_Basic_Acl` option (in the file `sdbasic.cm`) specifies the directory containing the ACL configuration files for Basic Security. This directory typically contains:

- `users.cfg` — the user configuration file
- `groups.cfg` — the group configuration file
- `acl.cfg` — the permissions file

Sd_Basic_Acl_Timeout

Used for: `sdbasic.cm`
Type: Real Number
Default Value: 3600
Valid Values: Any positive real number

The `Sd_Basic_Acl_Timeout` option (in the file `sdbasic.cm`) specifies the number of seconds Basic Security caches the configuration information, specified in `Sd_Basic_Acl`, in memory. Once the cache has expired, the configuration information is re-read. When set to 0, the cache never expires and is not re-read until the RTserver is restarted.

Sd_Basic_Admin_Msg_Types

Used for: `sdbasic.cm`
Type: String List
Default Value: -4, -30, -99
Valid Values: Integer values representing message type numbers, separated by commas

The `Sd_Basic_Admin_Msg_Types` option (in the file `sdbasic.cm`) specifies which message types require administrative privileges to publish. To grant administrative privileges to a user, place the user in the `admin` group. The `admin` group is defined in the `groups.cfg` configuration file.

Whenever an RTclient publishes a message of a message type specified by this option, the RTserver checks if the user has administrative privileges. If the user does not, the message is not routed.

The default message types requiring administrative privileges are CONTROL messages, SERVER_STOP_CALL messages, and GRP_STOP_RTGMS messages.



Authorizing permission for administrative messages is disabled when the `Authorize_Publish` option is set to `FALSE`.

Sd_Basic_Trace_File

Used for:	<code>sdbasic.cm</code>
Type:	String
Default Value:	<code>\$RTHOME/sdbasic.trc</code>
Valid Values:	Any valid filename, including <code>stdout</code> and <code>stderr</code>

The `Sd_Basic_Trace_File` option (in the file `sdbasic.cm`) specifies the name of the file to which Basic Security prints auditing information. Auditing information is specified by the `Sd_Basic_Trace_Level` option.

Sd_Basic_Trace_Flags

Used for:	<code>sdbasic.cm</code>
Type:	String List (Identifiers)
Default Value:	<code>prefix</code>
Valid Values:	<code>prefix</code> , <code>timestamp</code> , <code>unknown</code>

The `Sd_Basic_Trace_Flags` option (in the file `sdbasic.cm`) specifies how to format the security trace file (see `Sd_Basic_Trace_File` on page 556). If you specify `prefix`, the output prefix is included in the trace information. The prefix indicates the module from which the trace information originates. If you specify `timestamp` the trace information is timestamped.

You can specify either `prefix` or `timestamp` or both separated by a comma:

```
setopt sd_basic_trace_flags prefix, timestamp
```

Unsetting the option using either `unsetopt`, or using `setopt` to change the value to `unknown`, clears all the flags.

Sd_Basic_Trace_Level

Used for:	sdbasic.cm
Type:	String (Identifier) List
Default Value:	Info
Valid Values:	<ul style="list-style-type: none">• Never• Error• Warning• Info• Info_1• Info_2• Verbose• Verbose_1• Verbose_2• Debug

The `Sd_Basic_Trace_Level` option (in the file `sdbasic.cm`) specifies the amount of information Basic Security sends to the file specified by the `Sd_Basic_Trace_File` option. This information is useful for auditing purposes. At the `Warning` level, all failed authentication requests and denied authorization requests are printed. At the `Info` level, all authentication and authorization requests, both successful and unsuccessful, are printed.

Sender_Get_Reply

Used for:	RTclient processes only
Type:	Boolean
Default Value:	FALSE
Valid Values:	TRUE or FALSE

The `Sender_Get_Reply` option specifies whether or not a `TipcMsgGetSender` function call returns the sender field or the reply to message property. Setting this option to `TRUE` causes `TipcMsgGetSender` to return the reply to message property instead of returning the sender field.

Server_Async_Subscribe

Used for:	RTclient processes
Type:	Boolean
Default Value:	TRUE
Valid Values:	TRUE or FALSE

The `Server_Async_Subscribe` option specifies whether or not an RTclient waits for a response from RTserver after sending a new subscription request. Setting the option to `FALSE` causes a subscribing RTclient to wait for a confirmation from RTserver before processing messages. Setting `Server_Async_Subscribe` to `TRUE` makes the RTclient asynchronous, so that it does not block while waiting for a response from RTserver.

This option only affects subscription calls, such as `TipcSrvSubjectSetSubscribe`, from clients that are also using username and password or other authorization protocols to connect to RTserver. For example, clients using `TipcSrvSetCredentials` or `TipcSrvSetUsernamePassword` are affected. For all other clients, subscription requests are asynchronous by default.

This option is required and cannot be unset.



When `Server_Async_Subscribe` is `TRUE`, an RTclient is not immediately notified when a subscription request is authorized or denied. Unauthorized subscription requests are silently ignored by the RTserver. However, the RTclient updates internal subscription tables as the asynchronous subscription results come.

Server_Auto_Connect

Used for:	RTclient, RTmon, and RTgms processes
Type:	Boolean
Default Value:	TRUE
Valid Values:	TRUE or FALSE

The `Server_Auto_Connect` option specifies whether or not the RT process should automatically create a connection to RTserver if one is needed (for example, if RTclient tries to send a message to RTserver before it has created a connection to RTserver). If the RT process has a warm connection to RTserver, it can partially operate as if it still had a connection to RTserver (for example, outgoing messages are buffered). If `Server_Auto_Connect` is set to `FALSE` when this warm connection exists, the RT process does not automatically attempt to recreate a connection to RTserver.

This option is required and cannot be unset.

Server_Auto_Flush_Size

Used for:	RTclient and RTmon processes
Type:	Integer
Default Value:	8192
Valid Values:	Any integer 0 or greater

The `Server_Auto_Flush_Size` option specifies how many bytes of data are allowed to be buffered to be sent to RTserver before the data is automatically written, flushed, to the connection. If `Server_Auto_Flush_Size` is set to 0, all outgoing messages are automatically flushed immediately. If the RT process (RTclient or RTmon) is sending many messages in a short period of time, setting `Server_Auto_Flush_Size` to a larger value can lessen the amount of CPU time the RT process uses. This option is required and cannot be unset.

Server_Burst_Interval

Used for:	RTserver only
Type:	Real Number
Default Value:	0.5
Valid Values:	Any real number from 0.0 to 600.0, inclusive

The `Server_Burst_Interval` option specifies the burst interval in seconds used for a connection to an RTserver. When the RTserver has used up all its tokens on an RTserver connection (there are 0 tokens accumulated), the RTserver must wait for this burst interval before checking for more tokens to determine if it can send more data on its RTserver connection.

For more information, see the related options `Server_Max_Tokens` and `Server_Token_Rate`.

Server_Connect_Timeout

Used for:	RTserver only
Type:	Real Number
Default Value:	10.0
Valid Values:	Any real number greater than 0.0

The `Server_Connect_Timeout` option specifies the maximum amount of time (in seconds) RTserver waits when trying to read a `SRV_CONNECT_RESULT` initialization message from a new RTserver process (when two RTserver processes first rendezvous, they exchange `SRV_CONNECT_CALL` and `SRV_CONNECT_RESULT` initialization messages). If RTserver does not receive the message within the timeout period, RTserver destroys the connection to the new RTserver process.

This option is required and cannot be unset.

Server_Connection_Names

Used for: RTserver process only

Type: StringList

Default Value: UNKNOWN

Server_Connection_Names lets the server treat high-traffic client connections as if they were server connections—assigning them to the server thread pool. Specify these clients as a list of the connections' unique subject names. Wildcards are permitted.

For example, TIBCO SmartSockets Cache handles volume comparable to a server, but is not technically an RTserver. To assign it to the server thread pool, specify its name in this option.



For important background information, see Multi-Thread Mode on page 512.

This option is one of four that replace the deprecated option Server_Num_Threads:

- Multi_Threaded_Mode
- Server_Threads
- Client_Threads
- Server_Connection_Names

Server_Delivery_Timeout

Used for:	RTclient, RTmon, and RTgms processes
Type:	Real Number
Default Value:	30.0
Valid Values:	Any real number 0.0 or greater

The `Server_Delivery_Timeout` option specifies the default maximum amount of time (in seconds) to allow all receiving processes to acknowledge delivery of a guaranteed message. This default can be overridden by explicitly setting the delivery timeout of the message. The sending process does not synchronously wait for delivery to complete, but instead checks periodically. Checking for delivery timeouts is disabled if `Server_Delivery_Timeout` is set to 0.0. If a guaranteed message sent to `RTserver` is not acknowledged within `Server_Delivery_Timeout` seconds, this is an error, and a `GMD_FAILURE` message is processed.

This option is required and cannot be unset.

Server_Disconnect_Mode

Used for:	RTclient, RTmon, and RTgms processes
Type:	String
Default Value:	gmd_failure
Valid Values:	<ul style="list-style-type: none"> • warm • gmd_failure • gmd_success

The `Server_Disconnect_Mode` option specifies the action RTserver should take when the RT process (RTclient, RTmon, or RTgms) disconnects from RTserver. There are three possible values:

<code>warm</code>	RTserver saves subject information about RTclient or RTmon for GMD so that no messages are lost.
<code>gmd_failure</code>	RTserver destroys all information about RTclient or RTmon and causes pending guaranteed message delivery to fail.
<code>gmd_success</code>	RTserver destroys all information about RTclient or RTmon and causes pending guaranteed message delivery to succeed.

Setting the option to `warm` is useful when RTclient or RTmon must run continuously and not lose any messages even if it crashes or accidentally terminates. In this mode, RTserver remembers the subjects subscribed to by the disconnecting RT process and buffers GMD messages. When an RTclient or RTmon process with the same value for the `Unique_Subject` option reconnects to RTserver, RTserver resends the guaranteed messages to that RT process. The maximum amount of time (in seconds) RT process has to reconnect is controlled by the option `Client_Reconnect_Timeout`. If the RT process does not reconnect to RTserver within `Client_Reconnect_Timeout` seconds, RTserver clears the GMD messages that have not been acknowledged by this RT process and sends a `GMD_NACK` message back to the sender of these messages.

Setting the option to `gmd_failure` is useful for short-lived operations. In this mode, RTserver clears the guaranteed messages that have not been acknowledged by this RT process and sends a `GMD_NACK` message back to the sender of these messages. This is also the correct setting if your RT process is RTgms, which does not support GMD and cannot receive GMD messages.

Setting the option to `gmd_success` is useful for short-lived operations or when RTclient or RTmon must exit cleanly without causing GMD failure in the sending process. In this mode, RTserver clears the unacknowledged GMD messages and sends a `GMD_ACK` message back to the sender of these messages.

This option is required and cannot be unset.

Server_Gmd_Dir_Name

Used for:	RTclient processes
Type:	String
Default Value:	UNKNOWN
Valid Values:	Any valid sub-directory name. It cannot be a pathname or contain slashes.

The `Server_Gmd_Dir_Name` option is a named option that specifies the name of the sub-directory used to store messages for file-based GMD. This sub-directory is created within the directory specified by the `IpC_Gmd_Directory` option. Messages sent with GMD are saved on disk, in this sub-directory, to be resent in the event of a delivery failure. File-based GMD is slower than memory-based GMD because of the time needed to write the messages to disk, but it is more reliable in situations like a power outage. The default for this option is `UNKNOWN` and results in the sub-directory being named after the RTclient's unique subject. Read and write access to the directory you specify is required. For more information, see `IpC_Gmd_Directory`.

This option is required if you are using multiple RTserver connections (multiple RTserver connections instead of a single global connection) that share the same unique subject. By default, the GMD sub-directory is named after the RTclient's unique subject. If an RTclient is using multiple connections with file-based GMD, and each connection is using the same unique subject, file conflicts in the GMD sub-directory occur. To avoid this, you must set the `Server_Gmd_Dir_Name` named option, using `setnopt`, to a different value for each connection.



Multiple RTserver connections can share the same unique subject only if each connection is connecting to a different RTserver cloud or if they have different projects.

Server_Keep_Alive_Timeout

Used for:	RTclient, RTserver, RTmon, and RTgms processes
Type:	Real Number
Default Value:	15.0
Valid Values:	Any real number 0.0 or greater

The `Server_Keep_Alive_Timeout` option specifies how long (in seconds) to wait when checking if a connection to an RTserver is still alive. This check is called a keep alive and occurs if more than `Server_Read_Timeout` seconds have elapsed since the RT process (RTclient, RTserver, RTgms, or RTmon) last read any data from that RTserver. Keep alives are disabled if `Server_Keep_Alive_Timeout` is set to 0.0.

If the keep alive fails, then the connection to that RTserver is destroyed and:

- if the connection had been between an RTclient, RTmon, or RTgms process and that RTserver, the RTclient, RTmon, or RTgms process attempts to create a new connection to that RTserver
- if the connection had been between an RTserver process and that RTserver, the RTserver process might attempt to create a new connection to that RTserver, depending on the RTserver process' setting for `Server_Reconnect_Interval` (see `Server_Reconnect_Interval` on page 572)

The larger the value you set for `Server_Keep_Alive_Timeout` for your RT process, the longer your RTprocess must wait to detect a possible RTserver failure. However, if you set too low a value for `Server_Keep_Alive_Timeout`, your RT process might think that it has lost its connection to an RTserver when that RTserver is merely very busy.

This option is required and cannot be unset.

Server_Max_Reconnect_Delay

Used for:	RTclient and RTmon processes
Type:	Real Number
Default Value:	30.0
Valid Values:	Any real number 0.0 or greater

The `Server_Max_Reconnect_Delay` option specifies the upper bound on a random delay introduced when an the RT process (RTclient or RTmon) has to reconnect to RTserver.

This option is useful when an RTserver with many clients fails and all of those RT processes are attempting to reconnect. The delay enhances total reconnect time by slightly staggering reconnect requests. Setting the option to zero disables the delay.

This option is required and cannot be unset.

Server_Max_Tokens

Used for:	RTserver only
Type:	Integer
Default Value:	0
Valid Values:	Any integer from 0 to 2147483647, inclusive

The `Server_Max_Tokens` option specifies the maximum number of tokens that can accumulate for connections to RTservers. The default value of 0 specifies that an unlimited number of tokens can accumulate.

For more information, see the related options `Server_Burst_Interval` and `Server_Token_Rate`.

Server_Msg_Send

Used for:	RTclient and RTmon processes
Type:	Boolean
Default Value:	TRUE
Valid Values:	TRUE or FALSE

The `Server_Msg_Send` option specifies whether or not the RT process (RTclient or RTmon) can send messages to RTserver. Some messages sent internally by the SmartSockets IPC library such as `SUBJECT_SET_SUBSCRIBE` messages are always sent regardless of the setting of `Server_Msg_Send`. This option is useful for backup processes that should receive messages from RTserver but not send any out. For those processes, set the value to `FALSE`.

This option is required and cannot be unset.

Server_Names

Used for:	RTclient, RTserver, RTmon, and RTgms processes
Type:	String List
Default Value:	<code>_node</code> for RTclient, RTmon, RTgms processes UNKNOWN for RTserver
Valid Values:	Any valid logical connection name or names, separated by commas (,), or for RTserver, also UNKNOWN

The `Server_Names` option specifies a list of logical connection names used to find and start an RTserver. Each logical connection name has either of these forms:

- *protocol:node:address*, which can be shortened to *protocol:node*, *protocol*, or *node*
- *pgm:node:unicast_protocol.address*, used when the logical connection is for multicast, used by RTclients to connect to an RTgms process
- *proxy:node:address@protocol:dest_node:dest_address*, which is used when the logical connection to RTserver is through a proxy server (for example, a web proxy server with HTTP CONNECT enabled).

For more information about valid protocols, nodes, and addresses, see Logical Connection Names for RT Processes on page 192. Note that the start prefix can only be used by an RTclient to start an RTserver. No other RT process can use it to start an RTserver, and an RTclient cannot use it to start any process other than an RTserver.

For an RTclient, RTmon, or RTgms process, this option is required and cannot be unset. An RTclient must establish a connection to either an RTserver or an RTgms. RTmon or RTgms processes must establish a connection to an RTserver. These connections are required to communicate with other SmartSockets processes.

RTgms creates multiple connections to the RTserver specified in `Server_Names`: a single control channel connection and one connection for each group that RTgms manages.

A logical connection name of `pgm:localhost` does not connect to an RTgms process. If you want to use `localhost`, you must also specify `TCP` in the logical connection name:

```
pgm:localhost:tcp
```

For an RTserver process, the logical connection names specified in `Server_Names` are used to find other RTserver processes. For an RTserver process, you can set `Server_Names` to `UNKNOWN`, the default setting, and your RTserver process does not attempt to find any other RTservers.

Server_Num_Threads



This option is deprecated in release 6.7. We have retained its old behavior for backward compatibility, however, setting it triggers a warning. This option will be removed in a future release.

Release 6.7 introduces four new options that supersede this option:

- Multi_Threaded_Mode
- Server_Threads
- Client_Threads
- Server_Connection_Names

If any of these are set, they override the value of Server_Num_Threads.

For important background information, see Multi-Thread Mode on page 512.

Used for: RTserver process only

Type: Integer

Default Value: 1

Valid Values: Any integer greater than 0

The Server_Num_Threads option specifies how many threads should be used in a multi-thread session with RTserver. The default value is 1, which is a single-thread model and is the default mode for RTserver. Although the server does not enforce a maximum value, we advise caution when using large values; larger values do not necessarily result in better performance.

The optimal number of threads depends on the operational parameters of your deployment, such as the number of processors, the subject namespace, message fan-out characteristics, message rate, disk I/O rate, compression, the number of RTserver-to-RTclient connections, and the number of RTserver-to-RTserver connections (cloud configuration).

We recommend that you empirically determine the optimal value of this option for your deployment. We suggest that you begin tuning with the values in Table 19, and adjust them based on the results of your testing. We caution that the optimal value of this option can change significantly when the operational parameters of your deployment vary; when they do change, we strongly recommend that you re-test to determine the best value for this option.

Table 19 *Server_num_threads*

Number of CPUs	Initial Number of Threads
2-3	1 Our testing has not shown any benefit to adjusting this parameter on computers with fewer than 4 processors.
4 or more	number of CPUs plus 1

To run in multi-thread mode, your RTserver must be licensed for the SmartSockets MP option for the RTserver and specify a number of threads greater than 1 for the `Server_Num_Threads` option. If you specify a number greater than 1 and you do not have a SmartSockets MP license for that RTserver, you receive a warning message when the RTserver is started and it is started in single-thread mode. The `Server_Num_Threads` option is not available on all platforms. If you specify a number greater than 1 on a platform where `Server_Num_Threads` is not available, you receive an warning message when the RTserver is started; RTserver then starts in single-thread mode.

Server_Read_Timeout

Used for:	RTclient, RTserver, RTmon, and RTgms processes
Type:	Real Number
Default Value:	30.0
Valid Values:	Any real number 0.0 or greater

The `Server_Read_Timeout` option specifies how often (in seconds) data is expected to be available for reading on a connection to an RTserver. This timeout is used to check for possible network failures. If a read timeout occurs, a message is sent to RTserver to check if the connection is still alive. This check is called a keep alive. Checking for read timeouts is disabled if `Server_Read_Timeout` is set to 0.0. The larger the value for `Server_Read_Timeout`, the longer the RT process (RTclient, RTserver, or RTmon) must wait to detect a possible RTserver failure. If `Server_Read_Timeout` is set too low, however, the RT process may mistakenly think that it has lost its connection to the RTserver when the RTserver is merely very busy. See [Helpful Read Timeout Tips](#) on page 521 for more information.

If the message delivery timeout property is set to a value smaller than the value for `Server_Read_Timeout`, under certain circumstances the actual delivery timeout is the value you set for `Server_Read_Timeout`. For more information, see [Delivery Timeout](#) on page 321.

This option is required and cannot be unset.

Server_Reconnect_Interval

Used for:	RTserver only
Type:	Real Number
Default Value:	30.0
Valid Values:	Any real number greater than 0.0

The `Server_Reconnect_Interval` option specifies the interval (in seconds) at which RTserver attempts to reconnect to other RTserver processes listed in its `Server_Names` option. Reconnects also are attempted regularly for those RTservers that cannot be initially connected to (for example, because the other RTserver is not currently running). To prevent temporary deadlock, if two RTservers lose their connection to each other, only the RTserver that initially connected to the other attempts the reconnect. The RTserver attempts to reconnect immediately and then once again every `Server_Reconnect_Interval` seconds. If `Server_Reconnect_Interval` is set to 0.0, automatic reconnect is disabled.

This option is required and cannot be unset.

Server_Start_Delay

Used for:	RTclient, RTmon, and RTgms processes
Type:	Real Number
Default Value:	1.0
Valid Values:	Any real number greater than 0.0

The `Server_Start_Delay` option specifies how long (in seconds) the RT process sleeps between traversals of the list of connection names in the `Server_Names` option. This option, if used by multiple RTclients in conjunction with `Server_Start_Timeout` and `Server_Start_Max_Tries`, can also control the number of processes that attempt to start RTserver and the rate that these attempts occur.

If the RT process is using the `start_never` start prefix, the process still sleeps between traversals, but no automatic starts are preformed. For more information on start prefixes, see [Start Prefix](#) on page 195. The number of times the RT process traverses the connection names in the `Server_Names` option is dictated by the `Server_Start_Max_Tries` option.

This option is required and cannot be unset.

Server_Start_Max_Tries

Used for: RTclient and RTmon processes

Type: Integer

Default Value: 1

Valid Values: Any integer greater than 0

The Server_Start_Max_Tries option specifies how many times the RT process should traverse the connection names in the Server_Names option when attempting to find and/or start an RTserver. The RT process can not communicate with other SmartSockets processes if it cannot create a connection to RTserver.

This option is required and cannot be unset.

Server_Start_Timeout

Used for: RTclient and RTmon processes

Type: Real Number

Default Value: 30.0

Valid Values: Any real number greater than 0.0

The Server_Start_Timeout option specifies the maximum amount of time (in seconds) the RT process (RTclient or RTmon) waits for RTserver to finish initializing once that RT process has started RTserver. If RTserver does not finish initializing within the timeout period, the RT process (RTclient or RTmon) tries the next connection name in the Server_Names option.

This option is required and cannot be unset.

Server_Threads

Used for:	RTserver process only
Type:	Integer
Default Value:	0
Valid Values:	Any non-negative integer

Server_Threads determines the number of threads in the pool for connections from servers and high-traffic clients (see Server_Connection_Names on page 561).
For optimal load-balancing results, set this option to either the number of servers, or to an integer divisor of that number.



For important background information and the semantics of special values, see Multi-Thread Mode on page 512.

This option is one of four that replace the deprecated option Server_Num_Threads:

- Multi_Threaded_Mode
- Server_Threads
- Client_Threads
- Server_Connection_Names

Server-Token_Rate

Used for:	RTserver only
Type:	Integer
Default Value:	0
Valid Values:	Any integer from 0 to 2147483647, inclusive

The Server-Token_Rate option specifies the rate, in bytes a second, at which tokens accumulate for RTserver connections. Specifying a value of 0 disables server bandwidth control.

For more information, see the related options Server_Burst_Interval and Server_Max_Tokens.

Server_Write_Timeout

Used for:	RTclient, RTmon, and RTgms processes
Type:	Real Number
Default Value:	30.0
Valid Values:	Any real number 0.0 or greater

The `Server_Write_Timeout` option specifies how often (in seconds) data is expected to be able to be written to the connection to RTserver. This timeout is used to check for possible network failures. If a write timeout occurs, then the connection to RTserver is destroyed, and RTclient, RTmon, or RTgms attempts to create a new connection to RTserver. Checking for write timeouts is disabled if `Server_Write_Timeout` is set to 0.0. The larger the value for `Server_Write_Timeout`, the longer the RT process must wait to detect a possible RTserver failure. If `Server_Write_Timeout` is set too low, however, the RT process may think that it has lost its connection to RTserver when the RTserver is merely very busy.

This option is required and cannot be unset.

Sm_Security_Driver

Used for:	RTserver
Type:	String
Default Value:	None
Valid Values:	Any valid SmartSockets security driver. The only driver supported is the <code>basic</code> security driver.

The `Sm_Security_Driver` option specifies which security driver to load. If the specified security driver cannot be loaded, no connections are allowed. Currently, the only available security driver is `basic`.

Socket_Connect_Timeout

Used for:	RTserver, RTclient, RTmon, RTgms processes
Type:	Real Number
Default Value:	5.0
Valid Values:	Any real number 0.0 or greater

The `Socket_Connect_Timeout` option specifies the maximum amount of time (in seconds) the RT process waits when trying to create a client socket connected to a server process. This timeout is used to check for possible network failures. Checking for connect timeouts is disabled if `Socket_Connect_Timeout` is set to 0.0. All SmartSockets standard processes use sockets for interprocess communication.

This option is required and cannot be unset.

Srv_Client_Names_Min_Msgs

Used for:	RTserver
Type:	Boolean
Default Value:	FALSE
Valid Values:	TRUE or FALSE

The `Srv_Client_Names_Min_Msgs` option specifies the level of detail returned by RTserver in subsequent `MON_CLIENT_NAMES_STATUS` messages. With this option set to `TRUE`, RTserver constructs a `MON_CLIENT_NAMES_STATUS` message containing only the client name that was created or destroyed and the reason for the client's disconnection.

Srv_Subj_Names_Min_Msgs

Used for: RTserver
Type: Boolean
Default Value: FALSE
Valid Values: TRUE or FALSE

The `Srv_Subj_Names_Min_Msgs` option specifies the level of detail returned by RTserver in subsequent `MON_SUBJECT_NAMES_STATUS` messages. With this option set to `TRUE`, RTserver constructs a `MON_SUBJECT_NAMES_STATUS` message containing only the created or destroyed subject.

Subjects

Used for: RTclient and RTmon processes
Type: String List
Default Value: None
Valid Values: Any valid subject name or names, separated by commas (,)

The `Subjects` option lists the subjects to which the RTclient initially subscribes. One or more subjects can be listed, separated by commas. For example:

```
setopt subjects /system/eps, /system/pcs, /control/...
```

When using the `Subjects` option, you must call the function `TipcSrvStdSubjectSetSubscribe` to parse the `Subjects` option and automatically subscribe to all listed subjects.

`TipcSrvStdSubjectSetSubscribe` is used to start or stop subscribing to subjects listed in the `Subjects` option, including standard subjects such as `_all`, `_process`, and `_node`. When called, `TipcSrvStdSubjectSetSubscribe` parses the `Subjects` option so that all subjects listed in `Subjects` are automatically subscribed to or unsubscribed from. For more information on `TipcSrvStdSubjectSetSubscribe`, see the *TIBCO SmartSockets Application Programming Interface*. For more information on standard subjects, see `Standard Subjects` on page 162.

The RTclient can also start or stop subscribing to a subject at any time using the `subscribe` and `unsubscribe` commands.

Time_Format

Used for:	RTserver, RTclient, RTmon, and RTgms processes
Type:	String (Identifier)
Default Value:	<ul style="list-style-type: none">• hms for RTserver and RTmon processes• unknown for all other RT processes
Valid Values:	<ul style="list-style-type: none">• full• fullzone• hms• numeric• The name of any time converter defined by the user

The `Time_Format` option controls how the RT process displays the value of time. Four standard formats and one user-defined format are available:

<code>full</code>	displays a combination of the date and the time.
<code>fullzone</code>	displays a combination of the date and the time with a time zone designation
<code>hms</code>	displays the time in hours, minutes, and seconds.
<code>numeric</code>	displays the floating point representation of time.
<code>user_defined</code>	uses a user-defined time converter. For a complete description of a user-defined time converter, see the function <code>TutTimeCvtCreate</code> in the <i>TIBCO SmartSockets Utilities</i> reference.

This option is required and cannot be unset.

Trace_File

Used for:	RTserver, RTclient, and RTmon processes
Type:	String (Identifier)
Default Value:	unknown
Valid Values:	Any valid filename, including <code>stdout</code> and <code>stderr</code>

The `Trace_File` option specifies the name of a file to which the RT process should write trace information. This allows you to specify a file other than `stdout`. If the option is not set, and uses the default value of `unknown`, that causes the output to be written to `stdout`.

Trace_File_Size

Used for:	RTserver, RTclient, and RTmon processes
Type:	Integer
Default Value:	0
Valid Values:	Any integer 0 or greater

The `Trace_File_Size` option specifies the maximum size, in bytes, of a trace file. Once a trace file reaches this maximum size it is backed up according to the rules specified by the `Backup_Name` option, and reopened at the beginning. When the trace file reaches its maximum size again, the backup file is overwritten and the trace file reopens at the beginning. Setting `Trace_File_Size` to 0 disables the option, allowing a trace file to grow indefinitely.

This option is used to prevent a long-lived process from filling up the physical disk of the machine on which it is running with its trace output.

This option is required and cannot be unset.

Trace_Flags

Used for:	RTserver, RTclient, and RTmon processes
Type:	String List (Identifiers)
Default Value:	<code>prefix</code>
Valid Values:	<code>prefix</code> , <code>timestamp</code> , <code>unknown</code>

The `Trace_Flags` option specifies how to format the trace file. If you specify `prefix`, the output prefix is included in the trace information. The prefix indicates from which module the trace information originated. If you specify `timestamp` the trace information is timestamped.

You can specify either `prefix` or `timestamp` or both separated by a comma:

```
setopt trace_flags prefix, timestamp
```

Unsetting the option using either `unsetopt`, or using `setopt` to change the value to `unknown`, clears all the flags.

Trace_Level

Used for:	RTserver, RTclient, and RTmon processes
Type:	String (Identifier)
Default Value:	unknown
Valid Values:	<ul style="list-style-type: none">• Never• Error• Warning• Info• Info_1• Info_2• Verbose• Verbose_1• Verbose_2• Debug

The `Trace_Level` option determines the amount of trace information output by an RT process to the trace file. You can specify how much information, if any, is output. If the option is not set, and uses the default value `unknown`, the level of information output is `Warning`.

If you are running the `MP` option with multiple threads, the `Info_1` trace level is the first level to provide detailed thread information.

Udp_Broadcast_Timeout

Used for:	RTserver, RTclient, and RTmon processes
Type:	Real Number
Default Value:	5.0
Valid Values:	Any real number greater than 0.0

The `Udp_Broadcast_Timeout` option specifies the maximum amount of time in seconds the RT process waits for an RTserver to respond to its broadcast attempt to find a running RTserver. If no RTserver responds within the timeout period, the RT process tries the next connection name in the `Server_Names` option.

This option is required and cannot be unset.

Unique_Subject

Used for:	RTserver, RTclient, RTmon, RTgms processes
Type:	String
Default Value:	<i>_Node_Pid</i>
Valid Values:	Any valid string unique for all the RT processes to which a particular RTserver is connected

For RTclient, RTmon, and RTgms processes, the Unique_Subject option specifies a unique subject that RTclient, RTmon, or RTgms automatically subscribes to when it creates a connection to an RTserver. For RTserver processes, the Unique_Subject option specifies a unique name that RTserver uses to identify itself when it connects to other RTservers. For an RTserver, unlike for the other RT processes, the value for Unique_Subject is used as a destination only for CONTROL messages, not for other types of messages.

RTserver does not allow multiple RT processes (RTservers, RTclients, RTmons, or RTgms) in the same project to have the same value for the option Unique_Subject.

The default value for Unique_Subject for any RT process is *_Node_Pid*, where:

Node is the network node name of the computer on which the RT process is running.

Pid is the operating system process identifier of the RT process.

This option is required and cannot be unset.

Verbose

Used for:	RTserver, RTclient, RTmon, RTgms processes
Type:	Boolean
Default Value:	FALSE
Valid Values:	TRUE or FALSE

The Verbose option takes a boolean value (TRUE or FALSE) and specifies the level of detail to output to the terminal in response to commands. If Verbose is set to TRUE, RTserver outputs more detailed or verbose information to the screen. This option is useful when debugging RTserver.

This option is required and cannot be unset.

Zero_Recv_Gmd_Failure

Used for:	RTserver only
Type:	Boolean
Default Value:	FALSE
Valid Values:	TRUE or FALSE

The Zero_Recv_Gmd_Failure option specifies how guaranteed message delivery should complete when there are no RTclient processes subscribing to the destination subject of the message. If Zero_Recv_Gmd_Failure is FALSE, delivery is considered to be successful and a GMD_ACK message is sent back to the sender of the message. If Zero_Recv_Gmd_Failure is TRUE, delivery is considered to have failed and a GMD_NACK message is sent back to the sender of the message.

This option is required and cannot be unset.

Chapter 9 **Command Reference**

This chapter describes the SmartSockets command language that can be used with the various SmartSockets processes: RTserver, RTclient, and RTmon. In general, SmartSockets commands are not case-sensitive. However, using all lower-case for the commands makes your applications more portable across different operating systems and hardware platforms.

Topics

- *RTserver Commands, page 584*
- *RTclient Commands, page 585*
- *RTmon Commands, page 587*
- *RTacl Commands, page 589*
- *Command Reference, page 590*

RTserver Commands

In the RTserver environment, run-time commands can be executed in these ways:

- you can add them to the startup command file, `rtserver.cm`, where they are read and executed during process initialization
- you can send a CONTROL message to the `_server` subject or to the unique subject of the destination RTserver

Supported RTserver Commands

The commands supported for an RTserver are:

- `alias` — create an alias for a command
- `cd` — change the current working directory
- `connect` — connect to other RTserver processes
- `disconnect` — disconnect from other RTserver processes
- `echo` — display text in the output window of the process
- `help` — display usage information about commands
- `helpopt` — display information about options
- `quit` — quit RTserver
- `setopt` — view or set the value of an option
- `sh` — execute a shell command
- `source` — read commands from a file
- `stats` — output CPU and memory usage statistics for RTserver
- `subscribe` — start subscribing to a subject or list the subjects being subscribed to
- `unalias` — delete an alias
- `unsetopt` — unset an RTserver option
- `unsubscribe` — stop subscribing to a subject

RTclient Commands

In the RTclient environment, run-time commands can be executed in several ways:

- you can add them to the startup command file for an RTclient, where they are read and executed during process initialization
- you can call the function `TutCommandParseStr` or `TutCommandParseTypedStr`
- you can send a `CONTROL` message to a subject to which RTclient is subscribed

Most supported RTclient commands are initialized by default. However, an RTclient process must call the API function `TipcInitCommands` to enable access to the `connect`, `disconnect`, `subscribe`, and `unsubscribe` commands. The commands created with `TipcInitCommands` cannot be used with the SmartSockets multiple connections API.

Supported RTclient Commands

The commands supported for an RTclient are:

- `alias` — create an alias for a command
- `cd` — change the current working directory
- `connect` — connect to RTserver
- `disconnect` — disconnect from RTserver
- `echo` — display text in the output window of the process
- `edit` — invoke a text editor to edit a file
- `help` — display usage information about commands
- `helpopt` — display usage information about options
- `setopt` — view or set the value of an option
- `setnopt` — view or set the value of a named option
- `sh` — execute a shell command
- `source` — read commands from a file
- `stats` — output CPU and memory usage statistics for RTclient
- `subscribe` — list active subjects or start subscribing to one or more subjects

- `unalias` — delete an alias
- `unsetopt` — unset an RTclient option
- `unsubscribe` — stop subscribing to one or more subjects

RTmon Commands

In the RTmon GDI, you can enter commands using any of these methods:

- the GDI pulldown menus
- the command interface
- through CONTROL messages

The use of the RTmon GDI is discussed in Chapter 6, Using RTmon.



The RTmon GDI has been deprecated and may be removed in a future release.

Commands are entered into the RTmon command interface by typing commands from the keyboard. The command interface is similar to a shell like the UNIX C shell, OpenVMS DCL, MVS TSO, or the Windows command prompt. You can:

- set options
- step through messages
- request monitoring information
- and many other tasks

Supported RTmon Commands

RTmon is an RTclient, so many of the supported commands are the same as for other RTclients. The commands supported for RTmon are:

- `alias` — create an alias for a command
- `cd` — change the current working directory
- `connect` — connect to RTserver
- `create` — create a message type
- `disconnect` — disconnect from RTserver
- `echo` — display text in the output window of the process
- `edit` — invoke a text editor to edit a file
- `help` — display usage information about commands
- `helpopt` — display information about options
- `history` — list the commands previously entered into RTmon

- `poll` — make a one-time request for monitoring information
- `quit` — quit RTmon
- `run` — process one or more messages
- `send` — send a message to RTserver for distribution
- `setopt` — view or set the value of an option
- `sh` — execute a shell command
- `source` — read commands from a file
- `stats` — output CPU and memory usage statistics for RTmon
- `subscribe` — list active subjects or start subscribing to one or more subjects
- `unalias` — delete an alias
- `unsetopt` — unset an RTmon option
- `unsubscribe` — stop subscribing to one or more subjects
- `unwatch` — stop watching monitoring information
- `watch` — display monitoring information whenever it changes

RTacl Commands

In the RTacl environment, run-time commands can be executed in these ways:

- you can add them to the startup command file for RTacl, where they are read and executed during process initialization
- you can enter them in the command interface

Supported RTacl Commands

The commands supported for RTacl are:

- `alias` — create an alias for a command
- `cd` — change the current working directory
- `credentials` — create file-based credentials
- `echo` — display text in the output window of the process
- `edit` — invoke a text editor to edit a file
- `evaluate` — evaluate a user's permissions
- `groups` — list the groups in the ACL
- `help` — display usage information about commands
- `helpopt` — display information about options
- `history` — list the commands previously entered into RTacl
- `load` — load ACL configuration
- `permissions` — list the permissions in the ACL
- `quit` — quit RTacl
- `setopt` — view or set the value of an option
- `sh` — execute a shell command
- `source` — read commands from a file
- `unalias` — delete an alias
- `unsetopt` — unset an RTacl option
- `users` — list the users in the ACL

Command Reference

A reference page is supplied for each command:

- **Name** — the name of the command
- **Synopsis** — shows command, keywords, order and type of arguments; optional arguments are enclosed in []
- **Description** — describes what the command does
- **Caution** — describes possible side effects
- **See Also** — reference to related commands
- **Examples** — shows at least one example of using the command

alias

Name `alias` — create an alias for a command

Synopsis **Supported for RTserver, RTclient, RTmon, RTgms, and RTacl:**

```
alias
alias name
alias name definition
```

Description The `alias` command creates a synonym for a command. When an alias is used as the first word of a command, its definition is used in place of *name*. An alias is similar to a UNIX C shell alias or an OpenVMS DCL symbol. Aliases are useful as shortcuts for frequently used commands.

If `alias` is called without any arguments, it lists all aliases. If it is called with only *name*, it lists the definition of that alias. If `alias` is called with both *name* and *definition* (which can be any number of words), then the alias *definition* is assigned to *name*.

Three aliases, `ls`, `sh`, and `exit`, come predefined in all the SmartSockets processes on UNIX, Windows, and OpenVMS.

`ls` is defined as:

UNIX:

```
sh ls
```

OpenVMS:

```
sh directory
```

Windows:

```
sh dir
```

`pwd` is defined as:

UNIX:

```
sh pwd
```

OpenVMS:

```
sh show default
```

Windows:

```
sh cd
```

exit is defined as:

```
quit force
```

Caution Alias substitution is only accomplished on the first word of a command.

Because commands are separated by semicolons, it can be tricky to create an alias consisting of several commands. The solution is to put double quotes around the entire alias definition.

Do not create recursive aliases. The command interface cannot detect such aliases. In particular, aliases of the form `alias foo foo or alias foo bar; alias bar foo` cause an infinite loop.

To prevent irreversible damage to your SmartSockets system, this command should not be specified with the `Enable_Control_Msgs` option without careful supervision.

See Also `unalias, sh`

Examples

```
PROMPT> alias a alias
Alias a installed
PROMPT> alias q quit force
Alias q installed
PROMPT> alias
a alias
exit quit force
ls sh ls
pwd sh pwd
q quit force
PROMPT> alias exit
quit force
PROMPT> alias reconnect "disconnect; connect" /* quotes around definition */
PROMPT> reconnect
Disconnecting (warm) from RTserver.
Attempting to reconnect to RTserver.
Connecting to project <rtworks> on <_node> RTserver.
Using local protocol.
Message from RTserver: Connection established.
Start subscribing to subject </_workstation_26483> again.
```

cd

Name	<code>cd</code> — change the current working directory
Synopsis	Supported for RTserver, RTclient, RTmon, RTgms, and RTacl: <code>cd</code> <code>cd <i>pathname</i></code>
Description	The <code>cd</code> command changes the current working directory. If <code>cd</code> is called with an argument, it changes to the directory <i>pathname</i> . If <code>cd</code> is called without an argument, it changes to the user's home directory.
Caution	On OpenVMS, Windows, and UNIX, the <code>cd</code> command calls <code>getenv("HOME")</code> to determine the user's home directory. On OpenVMS, this fails if the DEC C run-time library has not been fully initialized (for example, when embedding SmartSockets into an OpenVMS FORTRAN program). On Windows systems, this fails where the <code>HOME</code> environment variable does not exist.
See Also	<code>sh</code>
Examples	<p>UNIX Examples:</p> <pre>PROMPT> pwd sh command executing with string: pwd /home/ssuser/src/ipc PROMPT> cd .. Current directory changed to .. PROMPT> pwd sh command executing with string: pwd /home/ssuser/src</pre> <p>OpenVMS Examples:</p> <pre>PROMPT> sh show def sh command executing with string: show default WKST1\$DKA300:[DEMO.SS55.STANDARD] PROMPT> cd [-] Current directory changed to [-] PROMPT> pwd sh command executing with string: show default WKST1\$DKA300:[DEMO.SS55]</pre> <p>Windows Examples:</p> <pre>PROMPT> cd .. Current directory changed to .. PROMPT> ls sh command executing with string: dir Volume in drive is WORKSTATION1 Volume Serial Number is 000D-0000</pre>

```
Directory of D:\users\ss55
11/18/99  09:24a      <DIR>      .
11/18/99  09:24a      <DIR>      ..
11/18/99  09:24a      <DIR>      ss55
          3 Files(s)          0 bytes
                        836,960,256 bytes free
```

connect

Name	<code>connect</code> — connect to RTserver
Synopsis	<p>Supported for RTserver:</p> <pre>connect connect <i>server_conn_name</i></pre> <p>Supported for RTclient and RTmon:</p> <pre>connect connect warm</pre>
Description	<p>The <code>connect</code> command connects the process to an RTserver (if it is not already connected).</p> <p>For RTclient and RTmon processes, if <code>connect</code> is called without any arguments, then a full global connection to RTserver is created. A full global connection is the normal mode for them to connect to RTserver. The <code>connect warm</code> command creates a warm connection to RTserver. A warm connection is a subset of a full connection, and should normally be used only when an RTserver is temporarily unavailable. The <code>connect</code> command uses the function <code>TipcSrvCreate</code> to create the connection to RTserver. For more information on creating a connection to RTserver, see Creating a Connection to RTserver on page 189.</p>



It is possible for the `connect` command to be received and processed via a `CONTROL` message, such as:

```
disconnect; setopt server_names tcp:bar; connect
```

If this occurs, the command will be applied to the connection on which it was received.

For RTserver processes using `connect`, it connects RTserver to other RTserver processes (if it is not already connected). A set of interconnected RTserver processes is called a group. When RTserver first starts up, it executes an implicit `connect` command as part of initialization. If `connect` is called without any arguments, RTserver traverses the list of logical connection names in the `Server_Names` option and tries to connect to all RTservers listed there. See [Finding Other RTserver Processes](#) on page 293. If `connect` is called with a logical connection name, RTserver connects to another RTserver using that specific logical connection name.

An RTclient process must call the API function `TipcInitCommands` to enable access to the `connect` command.

Caution To prevent irreversible damage to your SmartSockets system, this command should not be specified with the `Enable_Control_Msgs` option without careful supervision.

For RTserver processes, the `connect` command does nothing if RTserver is already connected to other RTserver processes.

The `connect` command cannot be used in any of the `rtserver.cm` startup command files, as RTserver creates its specific commands (such as `connect`) after the startup command files have been executed. For RTserver processes, the `connect` command can be executed using CONTROL messages with a destination of `_server`.

The `connect` command acts on the global connection unless received via a CONTROL message, in which case the command is applied to the connection on which it was received.

See Also `disconnect`, `subscribe`, `unsubscribe`

Examples Here is an example of connecting to an RTserver:

```
PROMPT> connect
Connecting to project <tutorial> on <artimus> RTserver.
Using local protocol.
Message from RTserver: Connection established.
Start subscribing to subject </_workstation_10770>.
```

create

Name	<code>create</code> — create a message type
Synopsis	<p>Supported for RTmon only:</p> <pre>create msg_type <i>name number grammar</i> [delivery_mode <i>default_delivery_mode</i>] [delivery_timeout <i>default_delivery_timeout</i>] [lb_mode <i>default_lb_mode</i>] [priority <i>default_priority</i>] [user_prop <i>default_user_prop</i>]</pre>
Description	<p>The <code>create</code> command creates a message type in RTmon. Once a new message type is created, the <code>send</code> command can be used to construct and send messages of that type. The <code>create</code> command is useful for prototyping new user-defined message types.</p> <p>The message type <i>name</i> (an identifier) and <i>number</i> (an integer) must not already be in use by an existing message type. Message type names that start with an underscore are reserved for internal use by SmartSockets. Message type numbers less than zero are reserved for standard SmartSockets message types. The message type <i>grammar</i>, which describes the layout of the data fields in messages of this type, must be a double-quoted string. For more information on message types, see Message Types. All standard message types (such as <code>CONTROL</code>) are available by default to all SmartSockets processes, including RTmon.</p> <p>By default, the new message type has a delivery mode of <code>BEST_EFFORT</code> unless it is a JMS message type. JMS message types have a delivery mode of <code>ORDERED</code> by default. The default delivery mode of a message type can be overridden by specifying a delivery mode (options are <code>BEST_EFFORT</code>, <code>ORDERED</code>, <code>SOME</code>, and <code>ALL</code>) for the <code>create</code> command.</p> <p>By default, the new message type has a delivery timeout of <code>UNKNOWN</code>, which causes the value of the option <code>Server_Delivery_Timeout</code> to be used for the delivery timeout of messages of this type. This can be overridden by specifying a delivery timeout.</p> <p>By default, the new message type has a load balancing mode of <code>NONE</code>. This can be overridden by specifying a load balancing mode (options are <code>NONE</code>, <code>ROUND_ROBIN</code>, <code>WEIGHTED</code>, and <code>SORTED</code>) for the <code>create</code> command.</p> <p>By default, the new message type has a priority of <code>UNKNOWN</code>, which causes the value of the option <code>Default_Msg_Priority</code> to be used for the priority of messages of this type. This can be overridden by specifying a priority (which must be an integer between -32768 and 32767, inclusive) for the <code>create</code> command.</p> <p>By default, the new message type has a user-defined property of zero (0). This can be overridden by specifying a user-defined property.</p>

Caution Message type names are not case sensitive.

There is no way to destroy a message type from the command interface.

Message types that are generated with `create` are transient and are lost when RTmon is terminated. Create commands can be stored in startup command files to generate more permanent message types for RTmon.

See Also `send`

Examples This example creates a user-defined XYZ_COORD_DATA message type with a delivery mode of ALL (delivery is guaranteed to all receivers), starts subscribing to the /system/test subject, and publishes an XYZ_COORD_DATA message to the /system/test subject (that is, sends a message to itself and any other processes subscribing to the /system/test subject):

```
MON> create msg_type xyz_coord_data 1 "int4 /*x*/ int4 /*y*/ int4
/*z*/" delivery_mode all
Created message type <xyz_coord_data> successfully.
MON> subscribe /system/test
Start subscribing to subject /system/test.
MON> send xyz_coord_data /system/test 1 4 9
Sent xyz_coord_data message to /system/test subject.
MON> run 1 1 /* read and process one message, but wait at most one second for it */
/* Default action for unexpected messages is to print out the message. */
Received an unexpected message.
type = xyz_coord_data
sender = </workstation1.talarian.com_4971>
sending server = </workstation1.talarian.com_4982>
dest = </system/test>
max = 2048
size = 48
current = 0
read_only = false
priority = 0
delivery_mode = all
ref_count = 1
seq_num = 1
server_seq_num = 8
resend_mode = false
user_prop = 0
ack to = <client:local:workstation1.talarian.com:RTSERVER>
data (num_fields = 3):
    int4 1
    int4 4
    int4 9
Processed a xyz_coord_data message.
```


credentials

Name `credentials` — create file-based credentials

Synopsis **Supported for RTacI:**

```
credentials file_name
credentials file_name -basic username password
```

Description The `credentials` command creates file-based credentials for the Basic Security. The credentials, which are a username and password, are written to the *file_name* file. The password is encrypted before being written to the file.

To use file-based credentials, you must also make the file available to your process by setting the `Auth_Data_File` option.

Specifying just the filename, without the username and password, prints the contents of the credentials file.

See Also None

Examples This example writes basic credentials to a file named `my_credentials`:

```
ACL> credentials my_credentials -basic jdoe dk30djf
Successfully created credentials
```

This prints the contents of the credentials created in the first example:

```
ACL> credentials my_credentials

Credential (my_credentials):
  Identifier: 2 (Basic)
  Data:      username=jdoe
             password=672800004f598eeafafd5b14ef8ffb742ab349cb
  Data Length: 84
```

disconnect

Name	<code>disconnect</code> — disconnect from RTserver
Synopsis	<p>Supported for RTserver:</p> <pre>disconnect disconnect <i>server_name</i> disconnect <i>server_conn_name</i></pre> <p>Supported for RTclient and RTmon:</p> <pre>disconnect disconnect full</pre>
Description	<p>The <code>disconnect</code> command disconnects the process from an RTserver (if it is connected). The process no longer receives any messages from that RTserver until the <code>connect</code> command is issued again. For an RTserver process, it is part of a group of interconnected RTserver processes as long as it is connected to other RTservers. When it disconnects, it is still considered part of a group of one.</p> <p>For RTclient and RTmon processes, if <code>disconnect</code> is called without an argument, the process keeps a warm connection to RTserver so that the process can later reconnect to that RTserver and still receive and watch the same subjects. The <code>disconnect full</code> command fully disconnects the process from that RTserver. The process can then continue as if it had never been connected to that RTserver. The <code>disconnect</code> command uses the function <code>TipcSrvDestroy</code> to destroy the connection to that RTserver. For more information on disconnecting from RTserver, see Destroying the Connection to RTserver on page 202.</p> <p>For RTserver processes calling <code>disconnect</code>, if it is called without any arguments, the RTserver process disconnects from all RTservers to which it is connected. If <code>disconnect</code> is called with the name of an RTserver, the RTserver process disconnects from that specific RTserver. If <code>disconnect</code> is called with a logical connection name, the RTserver process disconnects from the RTserver that it connected to using that specific logical connection name.</p> <p>An RTclient process must call the API function <code>TipcInitCommands</code> to enable access to the <code>disconnect</code> command.</p>

Caution To prevent irreversible damage to your SmartSockets system, this command should not be specified with the `Enable_Control_Msgs` option without careful supervision.

When using the `disconnect` command with an RTserver process, note that it does nothing if that RTserver process is not connected to other RTservers.

The `disconnect` command cannot be used in any of the `rtserver.cm` startup command files, as RTserver creates its specific commands (such as `connect`) after the startup command files have been executed. For RTserver processes, the `connect` command can be executed using CONTROL messages with a destination of `_server`.

The `disconnect` command acts on the global connection unless received via a CONTROL message, in which case the command is applied to the connection on which it was received.

See Also `connect`

Examples Here is an example of disconnecting from an RTserver:

```
PROMPT> disconnect
Disconnecting (warm) from RTserver.
/* Reconnect and continue. */
PROMPT> connect
Attempting to reconnect to RTserver.
Connecting to project <rtworks> on <_node> RTserver.
Using local protocol.
Message from RTserver: Connection established.
Start subscribing to subject </_workstation_13060> again.
/* Fully disconnect from RTserver. */
PROMPT> disconnect full
Disconnecting (full) from RTserver.
```

echo

Name	echo — display text in the output window of the process
Synopsis	Supported for RTserver, RTclient, RTmon, RTgms, and RTacl: echo <i>argument ...</i>
Description	<p>The <code>echo</code> command writes the specified arguments (any type) in the command to the normal output window of the process. The output window is standard output (<code>stdout</code>).</p> <p>The <code>echo</code> command is useful for producing diagnostics in command files.</p>
Caution	None
See Also	None
Examples	<pre>PROMPT> echo "Hello world! This is a test!" Hello world! This is a test! PROMPT></pre>

edit

Name	<code>edit</code> — edit a file
Synopsis	Supported for RTclient, RTmon, RTgms, and RTacl: <code>edit filename</code>
Description	The <code>edit</code> command invokes the text editor specified by the <code>Editor</code> option and loads <i>filename</i> . The default editor for UNIX is <code>vi</code> . The default editor for OpenVMS is <code>edt</code> . The default editor for Windows is <code>Notepad</code> . You may choose any editor or command that is available on the system by entering its name as the <code>Editor</code> option.
Caution	You must have write privilege for the file being edited before you can save any changes made to that file.
See Also	<code>setopt</code>
Examples	<p>UNIX Examples:</p> <pre>PROMPT> setopt editor emacs Option editor set to "emacs". PROMPT> edit /home/ssuser/text.file Editing file /home/ssuser/text.file /* displays an emacs editor window containing the file text.file */ PROMPT> setopt editor cat Option editor set to "cat". PROMPT> edit sensor.msg Editing file sensor.msg /* sensor.msg outputs to the output window since the editor option was set to cat */</pre> <p>OpenVMS Example:</p> <pre>PROMPT> EDIT [SSUSER]TEXT.FILE /* starts an edt editor session containing the file text.file */</pre> <p>Windows Example:</p> <pre>PROMPT> edit C:\autoexec.bat /* starts a Notepad editor session containing the file autoexec.bat */</pre>

evaluate

Name `evaluate` — evaluate a user’s permissions

Synopsis **Supported for RTacI:**
`evaluate permission user host resource`

Description The `evaluate` command is provided as a development aid to determine a user’s permissions in the currently loaded ACL. Provide a user, host and resource, and the ACL is evaluated and reports whether the permission is allowed or denied. If a host name rather than an IP address is provided, a DNS lookup is performed on the host name to determine the actual IP address.

The values permitted for *permission* and *resource* are:

Permission	Resource
server	
client	project
membership	group
publish	subject
subscribe	subject

If the permissions are not evaluated as you expect them to be, set the `Trace_Level` option to `debug`. This results in verbose output and can help determine which permissions are being invoked.

See Also None

Examples This example evaluates whether the user `jdoe` has permissions to subscribe to the subject `/stock/tibx` from the hostname `neptune`:

```
ACL> evaluate subscribe jdoe neptune /stock/tibx
ALLOW subscribe - user=<jdoe> host=<10.105.200.204>
resource=</stock/tibx>
```

This example evaluates whether the user `jdoe` has permissions to publish to the subject `/stock/tibx` from the hostname `neptune`:

```
ACL> evaluate publish jdoe neptune /stock/tibx
DENY publish - user=<jdoe> host=<10.105.200.204>
resource=</stock/tibx>
```

groups

Name	<code>groups</code> — list the groups in the ACL
Synopsis	Supported for RTacl: <code>groups</code>
Description	The <code>groups</code> command lists the groups and their users in the currently loaded ACL.
See Also	None
Examples	This example lists the groups and their users in the ACL: <pre>ACL> groups Group: dev jdoe Group: admin (administrative privileges) admin jdoe</pre>

help

Name	help — display usage information about commands
Synopsis	Supported for RTserver, RTclient, RTmon, RTgms, and RTacl: help help <i>command</i>
Description	<p>The <code>help</code> command displays usage information about the SmartSockets commands. If <code>help</code> is called without any arguments, it displays a summary of all commands. If the argument is the name of a valid command, more detailed information about <i>command</i> is displayed.</p> <p>The information displayed is similar to the Synopsis and Description information in this reference.</p>
Caution	None
See Also	helpopt
Examples	<pre>PROMPT> help setopt setopt setopt <option> setopt <option> <value></pre> <p>The <code>setopt</code> command is used to view or set the value of an option. If called with <code><option></code>, <code>setopt</code> displays the name of <code><option></code> and its current value. If called with <code><option></code> and <code><value></code>, <code>setopt</code> sets <code><option></code> to the new value. If called without arguments, <code>setopt</code> displays all process options and their current values.</p>

helpopt

Name	helpopt — display usage information about options
Synopsis	Supported for RTserver, RTclient, RTmon, RTgms, and RTacl: helpopt helpopt <i>option</i>
Description	The helpopt command displays usage information about SmartSockets options. If called with the name of a valid option, helpopt displays detailed information about <i>option</i> . If called without an option, helpopt displays a one line description for all the options supported for the RT process from which you called helpopt.
Caution	None
See Also	help, setopt
Examples	PROMPT> helpopt backup_name Backup_Name specifies the extension given to a backup file created when a file is opened in write mode. This includes all files created in "book". The backup file has the same name as the existing file with the addition of the extension specified in this option. Type: String Default: "~" on UNIX and Windows Unknown on OpenVMS

history

Name	history — view the command interface history
Synopsis	Supported for RTmon, and RTacl: history history <i>n</i>
Description	The <code>history</code> command is used to display a list of all the previously entered commands. When the <code>history</code> command is followed by a number, <i>n</i> , it lists only the last <i>n</i> commands.
Caution	None
See Also	help
Examples	PROMPT> history 1: help connect 2: help alias 3: subscribe foo 4: history

load

Name	load — load the ACL configuration
Synopsis	Supported for RTacI: load <i>directory</i>
Description	<p>The <code>load</code> command reads the ACL configuration files located in the specified directory and validates the syntax. An ACL must be loaded before any other commands which require an ACL can be run. Calling the <code>load</code> command when an ACL has already been loaded overwrites the previous ACL.</p> <p>Setting the <code>Trace_Level</code> option to <code>debug</code> results in verbose output and can help debug syntax errors.</p>
See Also	None
Examples	<p>This example loads the ACL configuration files located in the default directory:</p> <pre>ACL> load \$RTHOME/acl Successfully loaded ACL <c:/Program Files/SmartSockets/mainline/acl></pre>

permissions

Name	permissions — list the permissions in the ACL
Synopsis	Supported for RTacI: permissions
Description	The permissions command lists the permissions in the currently loaded ACL.
See Also	None
Examples	<p>This example lists the permissions in the ACL:</p> <pre>ACL> permissions Permissions: server allow group admin 10.105. * client allow user * * trade_project membership deny user * * * subscribe deny user * * /... subscribe allow group admin 10.105. /admin/... subscribe allow user * * /trade/... publish deny user * * /... publish allow group admin 10.105. /admin/... publish allow user * * /trade/...</pre>

poll

Name `poll` — make a one-time request for monitoring information

Synopsis **Supported for RTmon only:**

```
poll client_buffer client
poll client_cb client
poll client_cpu client
poll client_ext client
poll client_general client
poll client_info client
poll client_msg_type_ex client msg_type
poll client_msg_type client msg_type
poll client_names
poll client_names_num
poll client_option client option
poll client_subject_ex client subject
poll client_subject client subject
poll client_subscribe client
poll client_subscribe_num client
poll client_time client
poll client_version client
poll project_names
poll server_buffer server process
poll server_cpu server
poll server_conn
poll server_general server
poll server_msg_traffic_ex server process
poll server_msg_traffic server process
poll server_names
poll server_option server option
poll server_route server dest_server
poll server_start_time server
poll server_time server
poll server_version server
poll subject_names
poll subject_subscribe subject
```

Description The `poll` command sends a message to request a snapshot of some current monitoring information. Depending on the type of poll performed, the request may go to one or all RTserver and RTclient processes. After a poll, the `run` command can be used to receive and display the poll responses. The `poll` command uses the `TipcMonTypePoll` API functions (such as `TipcMonProjectNamesPoll` for `poll project_names`) to send the request.

For all of the user-specifiable parameters (*client*, *server*, *subject*, and so on), either a single item can be specified by name, or @ (an at sign) can be used to specify all items that match Monitor_Scope or all items of that type (such as all RTclients or all subjects). For *client* (RTclient processes), *server* (RTserver processes), and *process* (RTclient or RTserver processes), the process is identified by its unique subject. See Specifying Items to be Monitored on page 363 for complete information on monitoring parameters.

Polls ending in *_ex* supersede corresponding older forms because the older forms can truncate the traffic data. (The older forms are retained for backward compatibility with clients older than release 6.7.)

The poll *client_buffer* command polls *client* for buffer information.

The poll *client_cb* command polls *client* for callback information.

The poll *client_cpu* command polls *client* for client CPU usage.

The poll *client_ext* command polls *client* for information set by the user (with TipcMonExtSet*).

The poll *client_info* command polls *client* for general information.

The poll *client_general* command polls *client* for general information.

The poll *client_msg_traffic* command polls *client* for message traffic information. (Clients older than release 6.7 might return truncated values. If this occurs, RTmon will print an INFO level trace message.)

The poll *client_msg_type_ex* command polls *client* for message traffic information. (The older form *client_msg_traffic* is retained only for backward compatibility.)

The poll *client_names* command polls RTserver for client (RTclient processes) names.

The poll *client_names_num* command polls RTserver for the number of clients that are currently connected.

The poll *client_option* command polls *client* for option information.

The poll *client_subject_ex* command polls *client* for subject information. (The older form *client_subject* is retained only for backward compatibility.)

The poll *client_subscribe* command polls RTserver for the subjects to which *client* is subscribing.

The poll *client_subscribe_num* command polls RTserver for the number of subjects to which *client* subscribes.

The poll *client_time* command polls *client* for time information.

The poll *client_version* command polls *client* for version information.

The `poll project_names` command polls RTserver for project names.

The `poll server_buffer` command polls *server* for information on its buffer to *process*.

The `poll server_conn` command polls RTserver for connection information about all RTservers.

The `poll server_cpu` command polls RTserver for server CPU utilization.

The `poll server_general` command polls *server* for general information.

The `poll server_msg_traffic_ex` command polls *server* for message traffic information. (The older form `server_msg_traffic` is retained only for backward compatibility.)

The `poll server_names` command polls RTserver for server (RTserver processes) names.

The `poll server_option` command polls *server* for option information.

The `poll server_route` command polls *server* for route information.

The `poll server_start_time` command polls RTserver for start time and elapsed time information.

The `poll server_time` command polls *server* for time information.

The `poll server_version` command polls RTserver for version information.

The `poll subject_names` command polls RTserver for subject names in the current project.

The `poll subject_subscribe` command polls RTserver for the RTclient processes that are subscribing to the *subject* subject.

Caution The RTmon GDI uses the `poll`, `watch`, and `unwatch` commands to control RTmon. Because the GDI needs complete control over these commands, these three commands are unavailable for interactive use when using the GDI (for example, the user cannot enter a `poll` command at the GDI `MON>` prompt). These three commands are always available when using the runtime RTmon.

Only one project at a time can be monitored by each RTmon process. Information in other projects is not visible. To monitor a different project, the option `Project` must be changed, and then RTmon must disconnect and reconnect to RTserver.

The `poll` command causes RTmon to connect to RTserver if it has not already done so.

See Also `run`, `watch`, `unwatch`

Examples

```

MON> poll client_names
Polled for client_names.
MON> run 1
POLL> Current Clients:
    /_workstation1_13806 [Runtime_RTmon: ssuser@workstation1]
        connected to RTserver /_workstation1_13746
Processed a mon_client_names_poll_result message.
MON> poll client_general /... /*poll all clients */
Polled for client_general.
MON> run 1
/* Notice that we have polled ourself. */
Processed a mon_client_general_poll_call message.
MON> run 1
/* Here's the response from ourself. There would be additional responses if any other clients were
running. */
POLL> General Info From Client /_workstation1_13806
    ident = Runtime_RTmon, node = workstation1, user = ssuser, pid =
1376
    arch = sun4_solaris, project = rtworks,
        RTserver name = /_workstation1_1376
    logical conn name to RTserver: local:workstation1:RTSERVER
    subject subscribe: /_all /_mon /_workstation1 /_workstation1_1376
    int format = BIG_ENDIAN, real format = IEEE
    current sbrk = 274456, delta sbrk = 159744
Processed a mon_client_general_poll_result message.

```


quit

Name	<code>quit</code> — quit the RT process
Synopsis	Supported for RTserver, RTmon, and RTacl: <code>quit</code> <code>quit force</code> Supported for RTgms: <code>quit force</code>
Description	The <code>quit</code> command causes the RT process to exit. If <code>quit force</code> is used, no confirmation is requested.
Caution	None
See Also	<code>disconnect</code>
Examples	<pre>PROMPT> quit Really quit? no PROMPT> quit force This process is now exiting.</pre>

run

Name	run — process one or more messages
Synopsis	Supported for RTmon: run run <i>num_msgs</i> [<i>timeout</i>]
Description	<p>The <code>run</code> command causes the RT process to process one or more messages. If <code>run</code> is called without any arguments, it processes messages indefinitely.</p> <p>For RTserver, the command <code>run numloops</code> (where <i>numloops</i> is an integer) causes RTserver to run through its processing loop the specified number of times. This loop waits for data to arrive from another process or for a process to connect, and then processes all available data and new connections.</p> <p>For RTmon, the command <code>run num_msgs</code> (where <i>num_msgs</i> is a positive integer) causes RTmon to run through <i>num_msgs</i> messages. By default, RTmon waits indefinitely for each message to arrive. This can be overridden by specifying a <i>timeout</i> (in seconds). If <i>timeout</i> is reached, the <code>run</code> command stops and returns control to the command interface. To process all messages that are available within <i>timeout</i> seconds, use a command such as <code>run 1000000 timeout</code>.</p>
Caution	<p>The <code>run</code> command cannot be used in <code>rtserver.cm</code> startup command files, it is only available to RTserver once RTserver is fully initialized and ready to run.</p> <p>The <code>run</code> command causes RTmon to connect to RTserver if it has not already done so.</p>
See Also	<code>poll</code> , <code>watch</code> , <code>unwatch</code>
Examples	<pre>MON> watch client_names Start watching client_names. MON> run 2 5 /* wait up to 5 seconds for each message */ WATCH> Current Clients: /_workstation1_27497 [Runtime_RTmon: ssuser@workstation1] connected to RTserver /_workstation1_8055 Processed a mon_client_names_status message. Timeout of 5 was reached. MON> watch subject_names Start watching subject_names. MON> run 1 WATCH> Current Subjects: /_all /_workstation1 /_workstation127497 /_mon Processed a mon_subject_names_status message.</pre>

send

Name	send — send a message to RTserver for distribution
Synopsis	Supported for RTmon only: send <i>msg_type dest field1 ...</i>
Description	The <code>send</code> command constructs a message and sends it to RTserver for distribution to all RTclient processes subscribing to the destination subject. The message is specified in RTworks message file format. The outgoing message is not buffered, but is flushed immediately to RTserver for distribution. For information on message format, see Message Files on page 64.
Caution	The <code>send</code> command causes RTmon to connect to RTserver if it has not already done so. To prevent irreversible damage to your SmartSockets system, this command should not be specified with the <code>Enable_Control_Msgs</code> option without careful supervision.
See Also	<code>create</code>
Examples	<pre> <i>/* Send a message we won't get. */</i> MON> send numeric_data eps x 1 Sent numeric_data message to /eps subject. MON> subscribe eps Start subscribing to subject /eps. <i>/* Send a message we will get. */</i> MON> send numeric_data eps { y 2 z 3 } Sent numeric_data message to /eps subject. MON> run 1 <i>/* Default action for unexpected messages is to print out the message. */</i> Received an unexpected message. type = numeric_data sender = </workstation1.talarian.com_5031> sending server = </workstation1.talarian.com_4982> dest = </eps> max = 2048 size = 64 current = 0 read_only = false priority = 0 delivery_mode = best_effort ref_count = 1 seq_num = 0 resend_mode = false user_prop = 0 </pre>

```
data (num_fields = 4):  
  str "y"  
  real8 2  
  str "z"  
  real8 3  
Processed a numeric_data message.
```

setopt

Name	setopt — view or set the value of an option
Synopsis	Supported for RTserver, RTclient, RTmon, RTgms, and RTacl: setopt setopt <i>option</i> setopt <i>option value</i>
Description	The <code>setopt</code> command is used to view or set the value of an option. If called with <i>option</i> and <i>value</i> , <code>setopt</code> sets <i>option</i> to the new value. If called with <i>option</i> only, <code>setopt</code> displays the option name and its current value. If called without arguments, <code>setopt</code> displays all RT process options and their current values.
Caution	<p>If an option's value is a list of items, the items must be separated with commas (<i>value1, value2, ...</i>).</p> <p>To prevent irreversible damage to your SmartSockets system, this command should not be specified with the <code>Enable_Control_Msgs</code> option without careful supervision.</p>
See Also	<code>helpopt</code> , <code>unsetopt</code>
Examples	<p>Within a command file, you can change the <code>Time_Format</code> option to change how your <code>RTclient</code> displays time, from the default of showing time in hours, minutes, and seconds to showing both the date and time:</p> <pre>setopt time_format full</pre>

setnopt

Name	setnopt — view or set the value of a named option
Synopsis	<p>Supported for RTclient:</p> <pre>setnopt <i>name</i> setnopt <i>name option</i> setnopt <i>name option value</i></pre>
Description	<p>The <code>setnopt</code> command is used to view or set the value of a named option. Named options allow a group of options and their values to be associated with an arbitrary name. You must specify that arbitrary name when using the <code>setnopt</code> command. If the name has not been associated with that option before, the <code>setnopt</code> command creates the association in addition to displaying or setting the option's value. The <code>setnopt</code> command can be called in these ways:</p> <ul style="list-style-type: none"><code>setnopt <i>name option value</i></code><p>When called with <i>option</i> and <i>value</i>, <code>setnopt</code> sets <i>option</i> to the new value and associates <i>option</i> with <i>name</i> if <i>option</i> has not been associated with <i>name</i> previously.</p><code>setnopt <i>name option</i></code><p>When called with <i>option</i> but no <i>value</i>, <code>setnopt</code> displays the option name and its current value and associates <i>option</i> with <i>name</i> if <i>option</i> has not been associated with <i>name</i> previously.</p><code>setnopt <i>name</i></code><p>When called without <i>option</i> or <i>value</i>, <code>setnopt</code> displays all RTclient options of the specified name and their current values.</p>
Caution	<p>If an option's value is a list of items, the items must be separated with commas (<i>value1, value2, ...</i>).</p> <p>To prevent irreversible damage to your SmartSockets system, this command should not be specified with the <code>Enable_Control_Msgs</code> option without careful supervision.</p>
See Also	<code>helpopt</code>

Examples Within a command file, two sets of named options are created:

```
/* Option values for name client_1 */  
setnopt client_1 unique_subject /trader_1  
setnopt client_1 server_names tcp:moe, tcp:larry, tcp:curly  
  
/* Option values for name client_2 */  
setnopt client_2 unique_subject /trader_2  
setnopt client_2 server_names tcp:moe, tcp:larry, tcp:curly
```

sh

Name	sh — execute a shell command
Synopsis	Supported for RTserver, RTclient, RTmon, RTgms, and RTacl: sh <i>arguments</i>
Description	The sh command executes a shell command. On UNIX this shell is the Bourne shell. On OpenVMS this is DCL. On Windows this is the shell that you run in the command prompt (DOS box). The <i>arguments</i> are passed to the C runtime function system, which in turn passes them to the shell.
Caution	There is no way on UNIX to specify which shell to use. To prevent irreversible damage to your SmartSockets system, this command should not be specified with the Enable_Control_Msgs option without careful supervision.
See Also	cd
Examples	UNIX Examples: PROMPT> sh pwd sh command executing with string: pwd /home/ssuser/demos/eps PROMPT> sh date sh command executing with string: date Wed Jul 28 13:13:26 PST 1999 OpenVMS Examples: PROMPT> sh show default sh command executing with string: show default WKSTI\$DKA300:[SSUSER.DEMOS.EPS] PROMPT> sh show time sh command executing with string: show time 28-JUL-1999 13:16:25 Windows Example: PROMPT> sh date sh command executing with string: date The current date is: Wed 07/28/1999 Enter the new date: (mm-dd-yy)

source

Name	source — read and process commands from a file
Synopsis	Supported for RTserver, RTclient, RTmon, RTgms, and RTacl: source <i>filename</i>
Description	The source command reads the RTprocess commands from a file.
Caution	To prevent irreversible damage to your SmartSockets system, this command should not be specified with the Enable_Control_Msgs option without careful supervision.
See Also	sh
Examples	UNIX Examples: <pre>PROMPT> sh cat aliases sh command executing with string: cat aliases alias q quit force alias reconnect "disconnect; connect" alias date sh date PROMPT> source aliases Executing source command with file aliases Alias q installed Alias reconnect installed Alias date installed PROMPT> alias exit quit force ls sh ls pwd sh pwd q quit force reconnect disconnect; connect PROMPT> date sh command executing with string: date Wed Jul 28 14:01:24 PST 1999</pre>

OpenVMS Examples:

```
PROMPT> sh type aliases
sh command executing with string: type aliases
alias q quit force
alias reconnect "disconnect; connect"
alias date sh show time
PROMPT> source aliases
Executing source command with file aliases
Alias q installed
Alias reconnect installed
Alias date installed
PROMPT> alias
exit    quit force
ls      sh directory
pwd     sh show default
q       quit force
reconnect    disconnect; connect
PROMPT> date
sh command executing with string: date
28-JUL-1999 13:16:25
```

Windows Examples:

```
PROMPT> sh type aliases
sh command executing with string: type aliases
alias q quit force
alias reconnect "disconnect; connect"
alias date sh date
PROMPT> source aliases
Executing source command with file aliases
Alias q installed
Alias reconnect installed
Alias date installed
PROMPT> alias
exit    quit force
ls      sh dir
pwd     sh cd
q       quit force
reconnect    disconnect; connect
PROMPT> date
sh command executing with string: date
The current date is: Wed 07/28/1999
Enter the new date: (mm-dd-yy)
```

stats

Name	<code>stats</code> — output CPU and memory usage for the RT process
Synopsis	Supported for RTserver, RTclient, and RTmon: <code>stats</code>
Description	<p>The <code>stats</code> command causes the RT process to output CPU and memory usage about itself. It returns the accumulated CPU time in milliseconds since the last call to <code>stats</code> was made. It uses the standard C function <code>times</code> to calculate the CPU usage.</p> <p>To get a realistic accounting, enter <code>stats</code> as the first command upon entering the RT process and then as needed to do the analysis.</p>
Caution	The <code>stats</code> command reports the accumulated CPU time since the last time the command was issued.
See Also	None
Examples	<pre>PROMPT> stats Total accumulated CPU time: 0.583 seconds Total frames processed: 1 Current sbrk address: 198592 /* Execute some other commands here */ PROMPT> stats Total accumulated CPU time: 0.750 seconds Total frames processed: 13 Current sbrk address: 198592 Differences since last stats command: CPU time, 0.167 seconds, wall time, 35.603 seconds Frame count: 0, Frame rate: 0 frames per second Sbrk address changed by 0 bytes.</pre>

subscribe

Name	<code>subscribe</code> — start subscribing to one or more subjects or list the current subscriptions
Synopsis	<p>Supported for RTserver:</p> <pre>subscribe subscribe <i>project subject</i></pre> <p>Supported for RTclient and RTmon:</p> <pre>subscribe subscribe [-load_balancing_off] <i>subject1 subject2 ...</i></pre>
Description	<p>The <code>subscribe</code> command adds the specified subjects to the list of subjects to which the RT process is already subscribing. If no subjects are specified, the RT process displays the names of the subjects it is currently subscribed to. The <code>subscribe</code> command is additive. Each time you subscribe to a subject, it is added to the list of subjects to which the RT process is currently subscribed.</p> <p>An RTserver <code>subscribe</code> is normally used with a wildcard subject name that matches a large number of RTclient subscribers. This reduces the amount of inter-RTserver dynamic message routing information exchanged. For RTserver, the <code>subscribe</code> command can be used interactively, but is most commonly used from a <code>rtserver.cm</code> startup command file.</p> <p>For RTclient and RTmon, the <code>subscribe</code> command takes a variable number of subjects. By default, subscribing to a subject allows RTclient or RTmon to be a load-balanced receiver if a message is published to the subject with load balancing. This can be overridden if the <code>-load_balancing_off</code> modifier is specified, which allows RTclient or RTmon to always receive the subject in addition to one of the load-balanced subscribers. For more information on load balancing, see Chapter 3, Publish-Subscribe.</p> <p>An RTclient process must call the API function <code>TipcInitCommands</code> to enable access to the <code>subscribe</code> command.</p>

Caution To prevent irreversible damage to your SmartSockets system, this command should not be specified with the `Enable_Control_Msgs` option without careful supervision.

For RTserver, the `subscribe` command should only be used for projects where you have limited bandwidth (for example, over a 56.6K modem) or on large-scale projects where there are many matching subjects that can be covered by a wildcard subject `subscribe` in RTserver.

If RTclient or RTmon is not connected to RTserver, it does so before subscribing.

The `subscribe` command acts on the global connection unless received via a `CONTROL` message, in which case the command is applied to the connection on which it was received.

See Also `unsubscribe`

Examples Here is an example of subscribing to subjects for RTclients:

```
PROMPT> subscribe _time
Start subscribing to subject /_time.
```

unalias

Name	unalias — delete a command alias
Synopsis	Supported for RTserver, RTclient, RTmon, RTgms, and RTacl: unalias <i>name</i>
Description	The unalias command deletes a command alias.
Caution	To prevent irreversible damage to your SmartSockets system, this command should not be specified with the Enable_Control_Msgs option without careful supervision.
See Also	alias
Examples	<pre>PROMPT> alias r "disconnect; connect" Alias r installed PROMPT> alias ls sh ls pwd sh pwd r disconnect; connect PROMPT> unalias r Unalias command executed for alias r PROMPT> alias ls sh ls pwd sh pwd</pre>

unsetopt

Name	unsetopt — unset an option
Synopsis	Supported for RTserver, RTclient, RTmon, RTgms, and RTacl: unsetopt <i>option</i>
Description	The unsetopt command unsets an option (sets its value UNKNOWN).
Caution	Not all options can be unset. To prevent irreversible damage to your SmartSockets system, this command should not be specified with the Enable_Control_Msgs option without careful supervision.
See Also	setopt
Examples	<pre>PROMPT> unsetopt default_protocols Option default_protocols unset PROMPT> setopt default_protocols default_protocols UNKNOWN</pre>

unsubscribe

Name	unsubscribe — stop subscribing to one or more subjects
Synopsis	<p>Supported for RTserver:</p> <pre>unsubscribe <i>project subject</i></pre> <p>Supported for RTclient and RTmon:</p> <pre>unsubscribe <i>subject1 subject2 ...</i></pre>
Description	<p>The <code>unsubscribe</code> command deletes the specified subjects from the list of subjects to which the RT process is subscribing.</p> <p>For RTclient and RTmon, the command takes a variable number of subjects.</p> <p>An RTclient process must call the API function <code>TipcInitCommands</code> to enable access to the <code>unsubscribe</code> command.</p>
Caution	<p>If a subject is specified that the RT process is not subscribing to, no action is taken for that subject.</p> <p>To prevent irreversible damage to your SmartSockets system, this command should not be specified with the <code>Enable_Control_Msgs</code> option without careful supervision.</p> <p>The <code>unsubscribe</code> command acts on the global connection unless received via a <code>CONTROL</code> message, in which case the command is applied to the connection on which it was received.</p>
See Also	<code>subscribe</code>
Examples	<pre>PROMPT> unsubscribe eps Stop subscribing to subject /eps. PROMPT> unsubscribe thermal pcs Stop subscribing to subject /thermal. Stop subscribing to subject /pcs.</pre>

unwatch

Name	unwatch — stop watching monitoring information
Synopsis	<p>Supported for RTmon only:</p> <pre> unwatch client_buffer <i>client</i> unwatch client_msg_recv <i>client msg_type</i> unwatch client_msg_send <i>client msg_type</i> unwatch client_names unwatch client_subscribe <i>client</i> unwatch client_time <i>client</i> unwatch project_names unwatch server_conn unwatch server_names unwatch subject_names unwatch subject_subscribe <i>subject</i> </pre>
Description	<p>The <code>unwatch</code> command is used to stop watching monitoring information. The <code>unwatch</code> command uses the <code>TipcMonTypeSetWatch</code> API functions to stop the watching. See <code>watch</code> on page 634 for more information on the RTmon watch categories.</p> <p>For all of the user-specifiable parameters (such as <i>client</i>, <i>server</i>, or <i>subject</i>), a single item can be specified by name or @ (an at sign) can be used to specify all items that match <code>Monitor_Scope</code> or all items of that type (such as all RTclients or all subjects). For <i>client</i> (RTclient processes), <i>server</i> (RTserver processes), and <i>process</i> (RTclient or RTserver processes), the process is identified by its unique subject. See <code>Specifying Items to be Monitored</code> on page 363 for complete information on monitoring parameters.</p>
Caution	<p>The RTmon GDI uses the <code>poll</code>, <code>watch</code>, and <code>unwatch</code> commands to control RTmon. Because the GDI needs complete control over these commands, these three commands are unavailable for interactive use (entering at the GDI <code>MON></code> prompt) when using the GDI. These three commands are always available when using the runtime RTmon.</p> <p>Only one project at a time can be monitored by each RTmon process. Information in other projects is not visible. To monitor a different project, the option <code>Project</code> must be changed, and then RTmon must disconnect and reconnect to RTserver.</p> <p>The <code>unwatch</code> command causes RTmon to connect to RTserver if it is not already connected. A warning message is printed if an <code>unwatch</code> is requested for a category not being watched.</p>
See Also	<code>watch</code>

Examples This example shows how watching is started and stopped for certain subjects:

```
MON> watch subject_names  
Start watching subject_names.  
MON> run  
WATCH> Current Subjects: /_all /_workstation1  
/_workstation1_27709/_mon  
^C*** Received interrupt from user. ***  
MON> unwatch subject_names  
Stop watching subject_names.
```

users

Name	<code>users</code> — list the users in the ACL
Synopsis	Supported for RTacl: <code>users</code>
Description	The <code>users</code> command lists the users in the currently loaded ACL.
See Also	None
Examples	This example lists the users in the ACL: <pre>ACL> users Users: admin (administrative privileges) jdoe anonymous</pre>

watch

Name	<code>watch</code> — display monitoring information whenever it changes
Synopsis	<p>Supported for RTmon only:</p> <pre> watch watch client_buffer <i>client</i> watch client_msg_recv <i>client msg_type</i> watch client_msg_send <i>client msg_type</i> watch client_names watch client_subscribe <i>client</i> watch client_time <i>client</i> watch project_names watch server_conn watch server_names watch subject_names watch subject_subscribe <i>subject</i> </pre>
Description	<p>The <code>watch</code> command turns on watching for monitoring information. Depending on the type of watch performed, the request may go to one or all RTserver and RTclient processes. Every time the watched information changes, RTmon receives updated information. The <code>run</code> command can be used to receive and display the watch responses. For most watch categories, when watching is turned on, RTmon also receives an initial status message so that RTmon can immediately display the current status. The <code>watch</code> command uses the <code>TipcMonTypeSetWatch</code> API functions to start the watching.</p> <p>For all of the user-specifiable parameters (<i>client</i>, <i>server</i>, <i>subject</i>), a single item can be specified by name or @ (at sign) can be used to specify all items that match <code>Monitor_Scope</code> or all items of that type (such as all RTclients or all subjects). For <i>client</i> (RTclient processes), <i>server</i> (RTserver processes), and <i>process</i> (RTclient or RTserver processes), the process is identified by its unique subject. See Specifying Items to be Monitored on page 363.</p> <p>When @ (at sign) or a wildcarded value is used for a user-specifiable parameter, both current and future items of that type are watched. For example, the command <code>watch client_subscribe /...</code> not only turns on watching in all current RTclient processes but also all future RTclient processes!</p> <p>If <code>watch</code> is called without any arguments, all the categories currently being watched are printed, one on each line.</p> <p>The <code>watch client_buffer</code> command watches buffer information in <i>client</i>.</p> <p>The <code>watch client_msg_recv</code> command watches messages of type <i>msg_type</i> being received in <i>client</i>.</p> <p>The <code>watch client_msg_send</code> command watches message of type <i>msg_type</i> being sent from <i>client</i>.</p>

The `watch client_names` command watches client (RTclient processes) names.

The `watch client_subscribe` command watches the subjects to which *client* is subscribing.

The `watch client_time` command watches time in *client*.

The `watch project_names` command watches project names.

The `watch server_conn` command watches connection information about all RTservers.

The `watch server_names` command watches server (RTserver processes) names.

The `watch subject_names` command watches subject names.

The `watch subject_subscribe` command watches the RTclient processes that are subscribing to the *subject* subject.

Caution The RTmon GDI uses the `poll`, `watch`, and `unwatch` commands to control RTmon. Because the GDI needs complete control over these commands, these three commands are unavailable for interactive use (entering at the GDI `MON>` prompt) when using the GDI. These three commands are always available, though, when using the runtime RTmon.

Only one project at a time can be monitored by each RTmon process. Information in other projects is not visible. To monitor a different project, the option `Project` must be changed, and then RTmon must disconnect and reconnect to RTserver.

The `watch` command causes RTmon to connect to RTserver if it has not already done so.

The `watch` command prints a warning message if a `watch` is requested for a category that is already being watched.

See Also `run`, `unwatch`

Examples

```
MON> watch client_subscribe /...
Start watching client_subscribe </...>.
MON> run 100000 5
WATCH> Current Subjects Being Subscribed to by Client
/_workstatoin1_27709:
  /_all /_workstation1 /_workstation1_27709 /_mon
Timeout of 5 was reached.
```


Chapter 10 Using Multicast

SmartSockets provides a multicast feature to further enhance the features and performance of SmartSockets. SmartSockets Multicast implements reliable multicast to take full advantage of its bandwidth optimization properties. SmartSockets Multicast is an efficient way of routing a message to multiple recipients. The SmartSockets Multicast enables messages to be multicast to RTclients. SmartSockets Multicast uses a protocol called PGM to route messages and an RT process called RTgms to handle the message routing. This chapter describes how to configure and use the RTgms process.

In addition to RTgms, there is a new Group_Names option for RTclients using multicast, described in Chapter 8, Options Reference. There are new PGM options for RTclients, described in Setting PGM Options on page 657.

There is also an extended logical connection name that allows the RTclient to connect to the RTgms process. To enable an RTclient to receive or send multicast messages, the RTclient simply connects to the RTgms process, instead of connecting to an RTserver. The extended logical connection name is described in Address for Multicast on page 194.

Topics

- *Multicast Requirements, page 639*
- *One-to-Many Communications Solution, page 640*
- *Features, page 641*
- *Architecture, page 642*
- *Multicast Deployment Guidelines, page 643*
- *RTgms Overview, page 644*
- *Bandwidth Management, page 647*
- *RTgms Options, page 650*
- *RTgms Options Summary, page 653*
- *Option Reference, page 656*

- *Setting PGM Options, page 657*
- *Starting and Stopping RTgms, page 668*
- *Interrupting RTgms, page 671*
- *Sending a Message using Multicast, page 671*
- *RTgms Commands, page 672*
- *Tailoring Your Multicast Deployment, page 673*

Multicast Requirements

To use multicast with SmartSockets, you must have a license for the SmartSockets Multicast feature, separate from your standard SmartSockets license. Contact your TIBCO sales representative for more information on purchasing the feature. See the *TIBCO SmartSockets Installation Guide* for information on adding the license to your license file.

Any RTservers that RTgms connects to must be at the same SmartSockets version level as the RTgms process. Any RTclients receiving multicast must be running with the SmartSockets Version 6.0 runtime libraries or higher. To use the multicast protocol, PGM, your network hardware, such as routers and switches, must be configured for multicast. See your network administrator about your network supporting multicast.

For a discussion of when to use multicast, instead of unicast publish-subscribe, see *When Should I Use Multicast?* on page 170. SmartSockets Multicast is inherently threaded and must be run on a platform that supports the SmartSockets thread model.

SmartSockets Multicast is a negative acknowledgement (NAK)-based, reliable IP multicast transport protocol for applications that require ordered or unordered, duplicate-free, multicast data delivery from multiple sources to multiple receivers.

SmartSockets Multicast provides a scalable and efficient way to simultaneously transmit large amounts of data to a group of receivers, from a single sender if need be, over existing LAN, WAN, and satellite networks. SmartSockets Multicast enables the development of new multicast applications, provides a high degree of scalability, and is geared towards network efficiency.

One-to-Many Communications Solution

The majority of network communications employ a one-to-one, or unicast transmission model, where data is sent point-to-point from one sender to one receiver. Using this method, sending a message to 1,000 different people requires the transmission of 1,000 separate copies of the same message over the network, causing scalability and congestion problems. As the number of network users continues to grow, and as content becomes increasingly media-rich, enterprises are faced with rapidly escalating data distribution costs and bandwidth requirements just to maintain existing service levels.

IP multicast, an efficient way of delivering one-to-many communications, addresses these issues by enabling a single sender to simultaneously stream large amounts of data to many receivers and do so effectively in real time. Instead of sending 1,000 messages to reach 1,000 people, only one message need be sent. The efficiencies and cost-savings are enormous. Multicast also facilitates one-to-many applications that are impractical with unicast applications such as video and audio conferencing, employee communications, live Web transmissions of multimedia training, and multi-user games.

Raw IP multicast, although efficient, lacks a reliability layer. SmartSockets Multicast provides that reliability. SmartSockets Multicast enables simplified, highly reliable and scalable one-to-many data multicasting over terrestrial and satellite networks using the industry standard Pragmatic General Multicast (PGM) protocol.

Features

SmartSockets Multicast provides a timely, flexible solution for all multicast applications with high reliability and scalability requirements. TIBCO SmartSockets Multicast includes these features:

- automatically adjusts and optimizes NAK production.
- allows customers to configure the allocation of bandwidth to ODATA and RDATA.
- facilitates distributed applications that are not feasible with unicast technology
- enables highly scalable and reliable one-to-many data multicasting over most existing networks
- scales applications to a large number of users without increasing network bandwidth or server requirements
- reduces data distribution costs and bandwidth requirements by eliminating redundant transmissions
- distributes rich data content including XML, video, audio and graphics files to multiple sites in a highly efficient manner
- TIBCO SmartSockets Multicast is equally well-suited to terrestrial or satellite environments and to any data type
- ensures reliable IP multicasting over satellite—uniquely adapted to provide simple, reliable and efficient content deployment over satellite, one of the most cost-effective and practical network environments for multicast applications
- provides alternative, flexible recovery tools to match any need
- improves network efficiency
- optimizes real-time data delivery through data stream recovery
- routers are utilized in a manner that helps to reduce NAK traffic
- no control traffic generated if no loss occurs
- support for Forward Error Correction (FEC)
- proactive FEC when loss is uniform
- reactive FEC for efficient random loss recovery
- delayed FEC to recover from heavy amounts of loss, that is, greater than 1 minute

- a buffering model that allows late join support
- support for aggregated NAKs
- bandwidth rate control which can be set for all group channels between an RTgms process and RTservers or dynamically for specified group channels

Architecture

SmartSockets Multicast is designed to be connection-less and efficient. The routers in a network assist the end-host nodes by selectively forwarding protocol information along the most efficient path. However, a PGM-capable router is not mandatory for a small number of receivers, less than 100.

In addition, TIBCO has developed a product to act as a PGM-capable Network Element (NE) for networks without PGM-capable routers in them. PGM is a negative acknowledgment (NAK)-based protocol where the receivers notify the sender only in the case of message loss, the beneficial side-effect being the preservation of bandwidth.

A PGM sender does not explicitly know the receivers of any particular piece of data—yet this is not a liability. In fact, with its loose group membership model, PGM is extremely well-suited for applications that need to leave and join multicast groups quickly and with a minimum of control messages (or none at all). This makes PGM an ideal choice for applications that have transient or mobile properties.

Multicast Deployment Guidelines

TIBCO multicast applications depend on the network layer to provide multicast connectivity between them. To assist you in the successful deployment of our applications, it is beneficial to be aware of any issues that might adversely affect your implementation of SmartSockets Multicast.

To begin with, it is important to highlight the context into which multicast connectivity finds itself in, in contrast to unicast connectivity. The nearly seamless flow of unicast packets within our local area networks and across the Internet backbone is often taken for granted. Unicast connectivity through the Internet is now nearly ubiquitous. Unfortunately, multicast connectivity through the Internet and through most corporate intranets is typically far more constrained at this point.

Four of the seven layers in the ISO model of networking are referenced here, for the purpose of discussing multicast deployment guidelines. For each layer, the functions performed by the layer and the equipment or software that can perform them are discussed. The following table summarizes this information by layer.

Layer	Functions Performed	Software Providing Function
Application/ Middleware	File transfer, messaging	SmartSockets
Transport	Reliable stream delivery	SmartSockets Multicast, TCP
Network	Unreliable datagram routing between LANs, NAK Suppression, Designated Local Repairer	Routers, "Layer 3" switches, "smart" switches, PGM Network Element
Physical	Define LAN boundaries	Hubs, NICs, OS drivers, "dumb" switches, modems, CSU/DSUs

It is important to note that each layer relies on the layers below it to provide a foundation upon which it can build new functions.

RTgms Overview

The role of the RTgms process is to manage multicast groups and route multicast messages accordingly. The RTgms process works together with the RTserver to route messages. The RTgms process is similar to an RTserver and is configured in similar ways. You can start and stop the RTgms with the `rtgms` command, similar to the `rtserver` command, and RTgms can be installed as a service on Windows just as an RTserver can. The RTgms process connects to an RTserver in the same way that any RTserver connects to another RTserver, using the logical connection names specified in its `Server_Names` option.

When an RTgms process connects to an RTserver, two types of channel connections are established:

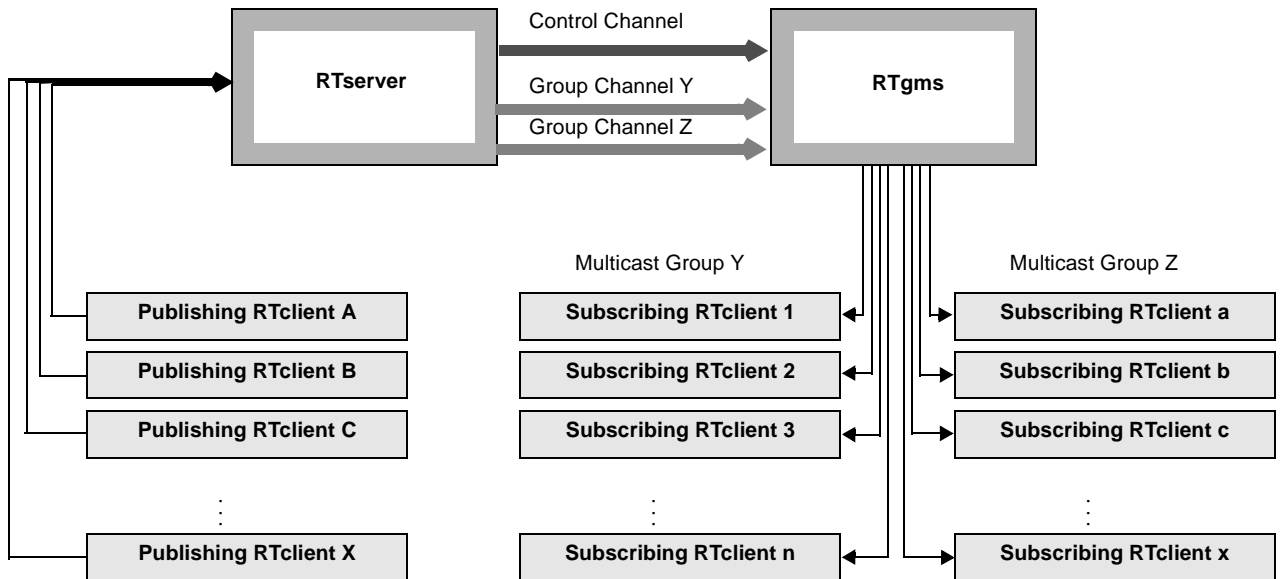
- group channel
- control channel

The group channel is the connection for sending messages to RTgms that are to be multicast to RTclients. There is a group channel for each multicast group that the RTgms manages.

The control channel is used for sending control messages from the RTserver to the RTgms. This enables the RTserver to treat the RTgms as an RTclient, and control messages can be used to monitor, stop, and dynamically reconfigure the RTgms process as if it were any other RTclient.

Figure 47 illustrates the message flow and connections in multicast messaging. Note that the connections on the left, between the publishing RTclients and the RTserver, are normal RTclient to RTserver connections. These are established using the logical connection name format *protocol:node:address*, such as `tcp:nodea`. The connections between the RTserver and RTgms, the control and group channels, are also normal RTserver to RTserver connections, using *protocol:node:address*, such as `tcp:nodea`. The connections between the subscribing RTclients and the RTgms use logical connection names that specify the PGM protocol. For example, they might use a logical connection name of `pgm:nodeb`.

Figure 47 Multicast Messaging



RTclients connect to an RTserver or an RTgms process. If an RTclient is connected to an RTserver and publishes a message for multicast, this is the message flow:

1. RTclient sends the message to the RTserver.
2. RTserver passes the message to the RTgms process.
3. RTgms checks the number of subscribing RTclients in each group connected to the RTgms.
4. If the number of subscribing RTclients in a group exceeds the `Group_Threshold` option, the RTgms process multicasts the message to those subscribing RTclients.

If the number of subscribing RTclients in a group is less than or equal to the `Group_Threshold` option, the RTgms process unicasts the message to those subscribing RTclients, using TCP/IP or local.

If the original publishing RTclient is connected to an RTgms process instead of an RTserver, the message passes through the RTgms process to the RTserver for routing. The RTserver always handles all routing. The RTserver then routes the message back to the RTgms process for multicasting.

The primary function of an RTclient, publish or subscribe, determines whether it should connect to an RTserver or an RTgms process. In general, RTclients that frequently publish messages for multicast should connect to an RTserver that is connected to the RTgms process. RTclients that frequently receive, or are subscribed to, multicast messages should connect to the RTgms process. This results in the most efficient message routing.

Bandwidth Management

When deploying an RTgms (SmartSockets with SmartPGM), there are two general approaches to rate control. The first approach, and the one recommended for SmartSockets customers, is to establish minimum and maximum rates, and assume RTclients will either keep up, or voluntarily drop out of the group. The second approach uses a technology within SmartPGM known as congestion control to automatically reduce the transmission rate to a level that can be handled by the slowest RTclient. Using congestion control is not generally recommended for SmartSockets customers who prioritize throughput over delivery.

Tuning Rate Control

By specifying the `pgm_source_max_trans_rate` and `pgm_source_min_trans_rate`, you can allocate the bandwidth necessary to send messages to subscribers promptly.

If you are unable to achieve the desired maximum rate, perform these steps:

1. Verify that congestion control is turned off.
2. Verify that the desired rate of the RTgms alone, or in combination with other applications on the network, does not exceed the bandwidth limitations of the network itself.
3. Verify that the RTgms is located on a machine with a sufficient amount of RAM to maintain the recovery window
(`pgm_source_transmit_size_buffer * pgm_source_max_trans_rate`)
without needing to use virtual memory. When the RTgms is forced to access the hard-disk to facilitate loss repair, performance degrades significantly.

Rate Control and Loss

It is common for a RTclient to experience small amounts of non-uniform (random) loss during its lifetime. When a RTclient experiences loss it will send a negative acknowledgement (NAK), unless configured otherwise. It is the responsibility of the RTgms to process NAKs. If the RTgms is unable to service a NAK, for whatever reason, the RTclient will suffer un-recoverable loss and the link between it and the RTgms will be gracefully terminated.



The most common cause of loss is setting the maximum send rate too high for the RTclients to sustain. If faced with persistent, unacceptable loss, first try reducing the value of the `pgm_source_max_trans_rate`.

Because SmartPGM is a reliable transport protocol, a RTgms makes recovering from loss the priority by default. When faced with loss, a RTgms will prioritize the transmission of repair data (RDATA) over the transmission of original data (ODATA). A consequence of loss in a single RTclient is the increased probability that latency will be introduced to all RTclients connected to same RTgms. As the amount of loss increases, the likelihood of latency also increases.



A misconfigured RTclient can suffer catastrophic loss (at or near 100% loss). In most cases a RTclient suffering catastrophic loss will ultimately suffer un-recoverable loss and disconnect. However, in the time prior to disconnect the effects of catastrophic loss on latency can be significant. It is recommended that a policy be established by the user to prevent misconfigured, or rogue RTclients, from arbitrarily connecting to a RTgms.

A RTgms can be configured to reduce the likelihood that latency will be introduced in the presence of loss.

`pgm_source_transmit_rdata_max_percentage` limits the bandwidth that an RTgms allocates for RDATA as a percent of `pgm_source_max_trans_rate`. `pgm_source_transmit_rdata_max_percentage` allows the user to shift the priority of SmartPGM from recovery to delivery.

Using `pgm_source_max_trans_rate` to shift the priority of SmartPGM from recovery to delivery increases the likelihood that a RTclient experiencing high loss will suffer un-recoverable loss and drop its connection to the RTgms. The decision to use `pgm_source_max_trans_rate` represents a trade-off between reliability and timely delivery.

Congestion Control

Using SmartPGM congestion control, RTgms regulates the transmission rate to accommodate the poorest performing RTclient in the multicast group. When an RTclient sends a negative acknowledgement (NAK), the RTgms reduces the transmission rate until the RTclient catches up and sends an acknowledgement (ACK). After receiving the ACK, the RTgms increases the rate to the specified maximum rate until the RTgms receives another NAK from an RTclient.

RTgms Options

RTgms options can be set to specific values by defining them in the `rtgms.cm` command file. Option values that have been specified in the command file are set each time RTgms is started, or they can be modified using the `setopt` command in a CONTROL message. For general information on setting options, see Setting Option Values on page 494. For information on conventions used in specifying options, see Specifying Options on page 497. For information on the RTgms command file, see RTgms Startup Command Files on page 652.

Certain options can be modified dynamically for a particular RTgms connection using an ADMIN_SET message. When you send the message to a particular RTgms process, the options apply only to outbound data sent on the specified group channel connection.

The ADMIN_SET message used for setting bandwidth rate control options is:

```
T_MT_GRP_ADMIN_SET_OUTBOUND_RATE_PARAMS
T_STR      group_name
T_INT4     token_rate
T_INT4     max_tokens
T_REAL8    burst_interval
```

where:

- group_name* must be the multicast group name of the group channel, matching an existing group name specified by the Group_Names option.
- token_rate* is the rate in bytes/second at which tokens accumulate. A value of -1 indicates no change. *token_rate* is equivalent to the Group_Token_Rate option.
- max_tokens* is the maximum number of tokens that can accumulate. A value of -1 indicates no change. *max_tokens* is equivalent to the Group_Max_Tokens option.
- burst_interval* is the burst interval in number of seconds. A value of -1.0 indicates no change. *burst_interval* is equivalent to the Group_Burst_Interval option.

The ADMIN_SET message used for setting PGM options is:

```
T_MT_GRP_ADMIN_SET_PGM_OPTIONS
T_STR    group_name
T_STR    option_name
T_INT4   option_value
```

where:

group_name is the name of the multicast group for which you want these parameters set. It needs to match an existing group name specified by the Group_Names option. The options apply to the group channel used by Group_Names.

option_name is the name of the PGM option you want to set. Currently, the only PGM option you can set in this way is Pgm_Source_Max_Trans_Rate. For *option_name*, the valid values are currently `source_max_trans_rate` and `pgm_source_max_trans_rate`.

option_value is the value you are specifying for *option_name*. For `Pgm_Source_Max_Trans_Rate`, the valid values are any integers greater than 0 and the default value is 4000000.

RTgms Startup Command Files

The RTgms process has standard startup command files, called `rtgms.cm`. The RTgms startup command file contains generic information that RTgms needs to know, such as what values to use for IPC-related timeouts, connection names, maximum number of clients, and so on.

This example illustrates a typical RTgms startup command file:

```
setopt project ss_test
setopt conn_names pgm:_node
setopt server_names tcp:_node
setopt group_threshold 10
setopt group_names rtworks, group1, group2
```

RTgms recognizes three levels of startup command files. When first invoked, it searches for and executes the commands in each file, in this order:

1. the system-level `rtgms.cm` file in the SmartSockets standard directory

RTgms searches for a system-level process command file `rtgms.cm` in the SmartSockets directory `standard`, in `RTHOME`:

- **UNIX:** `$RTHOME/standard`
- **Windows:** `%RTHOME%\standard`

To add or change RTgms options, change to this directory and use an editor to add or change the system-wide option settings.

2. the user-level `rtgms.cm` file in the user's home directory

RTgms searches for an `rtgms.cm` file in the user's home directory (specified by the `HOME` environment variable) and, if found, executes the commands in that file. This file is the ideal place to set options that you want set for all your projects. To create this file, use an editor to open a new file named `rtgms.cm` in your home directory:

- **UNIX:** `$HOME`
- **Windows:** `%HOME%`

and use the editor to add the options you choose.

3. the file specified by the `-command` argument, or the local-level `rtgms.cm` file in the current directory (if the `-command` argument is not specified)

RTgms reads and executes the `rtgms.cm` file found in the current directory, that is, the directory from which RTgms is being run. In this file, place any project-specific option declarations, such as the name of the project and where to find RTserver. The local command file is read last to allow you to override any initial values that might have been set for the RTgms options.

RTgms Options Summary

The table summarizes the relevant options available in all RTgms processes. These options can be modified using the `setopt` command in the RTgms startup command file.

Table 20 RTgms Options

Option Name	Type	Default
Backup_Name	String	~
Client_Connect_Timeout	Numeric	10.0
Client_Keep_Alive_Timeout	Numeric	0.0
Client_Max_Buffer	Numeric	10000000
Client_Read_Timeout	Numeric	0.0
Conn_Names	String List	UNIX: pgm:_node:local.5104, pgm:_node:tcp.5104 Windows: pgm:_node:tcp.5104
Default_Msg_Priority	Numeric	0
Default_Protocols	Identifier List	UNIX: local, tcp Windows: tcp
Default_Subject_Prefix	String	None
Editor	String	UNIX: vi Windows: notepad
Enable_Control_Msgs	String List	echo, quit
Group_Burst_Interval	Numeric	0.5
Group_Max_Buffer	Numeric	10000000
Group_Max_Tokens	Numeric	0

Table 20 RTgms Options

Option Name	Type	Default
Group_Names	String List	rtworks
Group_Threshold	Numeric	1
Group_Token_Rate	Numeric	0
IpC_Gmd_Directory	String	UNIX: /tmp/rtworks Windows: %TEMP%\rtworks
IpC_Gmd_Type	String	default
Log_In_Client	String	None
Log_In_Data	String	None
Log_In_Group	String	None
Log_In_Internal	String	None
Log_In_Status	String	None
Log_Out_Client	String	None
Log_Out_Data	String	None
Log_Out_Group	String	None
Log_Out_Internal	String	None
Log_Out_Status	String	None
Monitor_Scope	String	/*
Project	Identifier	rtworks
Real_Number_Format	String	%g
Server_Auto_Connect	Boolean	TRUE
Server_Delivery_Timeout	Numeric	30.0
Server_Disconnect_Mode	Identifier	gmd_failure

Table 20 RTgms Options

Option Name	Type	Default
Server_Keep_Alive_Timeout	Numeric	15.0
Server_Names	String List	_node
Server_Read_Timeout	Numeric	30.0
Server_Start_Delay	Numeric	1.0
Server_Write_Timeout	Numeric	30.0
Socket_Connect_Timeout	Numeric	5.0
Subjects	String List	None
Time_Format	Identifier	unknown
Unique_Subject	String	_Node_Pid
Verbose	Boolean	FALSE

Note that for the Conn_Names option, the default value is different for RTgms than it is for an RTserver. The `pgm` prefix results in RTgms using the PGM link driver. For example, in UNIX, the PGM link driver creates a server-side connection on TCP/IP, port 5104 on `_node` and also on local, port 5104 on `_node`. RTgms sends data out either on a multicast address (if the number of receivers meets the value set in the Group_Threshold option) or on the TCP/IP or local connection of the client.

Option Reference

Group_Threshold

Used for:	RTgms processes only
Type:	Integer
Default Value:	1
Valid Values:	Any integer greater than 0

The Group_Threshold option specifies the minimum number of clients required to send a message using multicast. The number of clients is per message per group. If the number of clients to receive a message is less than or equal to the value specified for Group_Threshold, the message is sent unicast using TCP/IP or local to that group. If the number of clients is greater than the value for Group_Threshold, the message is sent multicast to that group. If an RTgms is managing several groups, a message might be unicast to one group that didn't meet the group threshold, and multicast to other groups that do meet the threshold.

Setting PGM Options

The PGM options used for RTclients and RTgms processes are different from the other SmartSockets options. You can use the defaults for these options if the default PGM configuration works for you. If you use the defaults, you do not need to change or set any values because there are no required options.

However, if you need to change a value for either your RTclient or RTgms process, you must create a multicast command file, `mcast.cm`, for the RT process. The multicast command file is treated just like any startup command file, with the exception that there is only a system-level `mcast.cm` file. There are no user-level or local-level `mcast.cm` files. This system-level `mcast.cm` file is located in the SmartSockets standard directory or partitioned dataset:

- **UNIX:** `$RTHOME/standard`
- **Windows:** `%RTHOME%\standard`

In your `mcast.cm` file, set the PGM options you want to change using `setopt`. Here is a sample `mcast.cm` file:

```
setopt  pgm_source_max_trans_rate 5000000  /* change max trans rate to 5 meg
*/
setopt  pgm_udp_encapsulation      1        /* enable UDP encapsulation  */
```

If you want to share an `mcast.cm` file among several RTgms or RTclient processes, you can specify which `mcast.cm` file they should use in the `mcastopts.cm` file, also located in the `standard` directory. The `mcastopts.cm` file contains one option, `mcast_cm_file`.

For certain PGM options, you can set them dynamically for a particular group channel using the `T_MT_GRP_ADMIN_SET_PGM_OPTIONS` message. Currently, only the `Pgm_Source_Max_Trans_Rate` option can be set using this message. For more information, see RTgms Options.

mcast_cm_file

- Used for:** RTclient and RTgms processes
- Type:** String
- Default Value:** None
- Valid Values:** Any valid pathname, specified without % characters

The `mcast_cm_file` option specifies the fully qualified pathname to the `mcast.cm` file the RT process should use. You can use `setopt` to set the option, just as you do in your `mcast.cm` file.

Under Windows, if you specify an environment variable in the path, use a `$` and not `%` characters in the name. For example, use `$RTHOME`. Do not use `%RTHOME%`.

PGM Option Summary

The table summarizes the relevant PGM options available in all RTclient and RTgms processes:

Table 21 RTclient and RTgms PGM Options

Option Name	Type	Default
Pgm_Port	Numeric	5202
Pgm_Receive_Nak_Ttl	Numeric	1
Pgm_Receive_Pgmcc	Boolean	TRUE
Pgm_Receive_Pgmcc_Acker_Interval	Numeric	500
Pgm_Receive_Pgmcc_Loss_Constant	Numeric	60000
Pgm_Source_Admit_High	Numeric	10
Pgm_Source_Admit_Low	Numeric	5
Pgm_Source_Group_Ttl	Numeric	1
Pgm_Source_Max_Trans_Rate	Numeric	4000000
Pgm_Source_Min_Trans_Rate	Numeric	56000
Pgm_Source_Pgmcc	Boolean	TRUE

Table 21 RTclient and RTgms PGM Options

Option Name	Type	Default
Pgm_Source_Pgmcc_Acker_Selection_Constant	Numeric	75
Pgm_Source_Pgmcc_Init_Acker	String	0.0.0.0
Pgm_Udp_Encapsulation	Numeric	TRUE

Pgm_Port

- Used for:** RTgms processes only
- Type:** Integer
- Default Value:** 5202
- Valid Values:** Any valid and unique port number

The Pgm_Port option specifies the IP port on which the multicast packets are sent. This is the outgoing port, and is different from the port used for RTclients connecting to RTgms. This port is used for sending the multicast messages and other information. The port number you specify must be unique to the system. If an RTserver is running on the same machine, the value for Pgm_Port must be different than the port number used by the RTserver for the values specified in its Conn_Names option, which has a default of 5101.

Pgm_Receive_Nak_Ttl

Used for:	RTclient processes only
Type:	Integer
Default Value:	1
Valid Values:	Any integer between 1 and 255, inclusive

The `Pgm_Receive_Nak_Ttl` option specifies the initial time to live (TTL) for a NAK sent by an RTclient to an RTgms. This is one less than the maximum number of routers that a NAK can travel through before reaching the RTgms process.

NAKs are generated by an RTclient when data from the RTgms fails to reach the RTclient or arrives corrupted. When the RTgms receives a NAK, it generates repair data.

The time to live (TTL) for a NAK is automatically decremented every time the NAK passes through a router. The NAK is discarded by the router if, after it decrements the TTL, the TTL is 0. This allows a NAK stuck in a loop between routers to eventually be eliminated. Too small a value for `Pgm_Receive_Nak_Ttl` can cause the NAK to be discarded too soon, before it ever reaches an RTgms. The RTgms never gets a chance to generate repair data and the RTclient that generated the NAK can have unrecoverable losses. Too large a value for `Pgm_Receive_Nak_Ttl` can increase the amount of congestion caused by routing loops.

The default value of 1 only works if a NAK does not pass through any routers to reach an RTgms. Consider the number of routers a NAK must pass through to reach an RTgms, especially if the RTgms is on another network, when setting the value for `Pgm_Receive_Nak_Ttl`.

Pgm_Receive_Pgmcc

Used for:	RTclient processes only
Type:	Boolean
Default Value:	TRUE
Valid Values:	TRUE or FALSE

The Pgm_Receive_Pgmcc option specifies whether this RTclient can be a potential ACKer. The default, TRUE, specifies that this RTclient can be an ACKer. Setting this option to FALSE prevents this RTclient from ever being an ACKer, even if it is the first to NAK.

Pgm_Receive_Pgmcc_Acker_Interval

Used for:	RTclient processes only
Type:	Integer
Default Value:	500
Valid Values:	Any positive integer

The Pgm_Receive_Pgmcc_Acker_Interval option specifies the timeout in milliseconds for the initial ACKer. This timeout applies regardless of how the initial ACKer was chosen, either specified as initial ACKer using the Pgm_Source_Pgmcc_Init_Acker option or chosen as initial ACKer because it was the first to NAK.

Pgm_Receive_Pgmcc_Loss_Constant

Used for:	RTclient processes only
Type:	Integer
Default Value:	60000
Valid Values:	Any integer between 0 and 65536, inclusive

The Pgm_Receive_Pgmcc_Loss_Constant option specifies how the loss rate is measured. The value you specify for this option is divided by 65536 to determine the percentage of gain used in the low pass filter for loss measurement.

Pgm_Source_Admit_High

Used for:	RTgms processes only
Type:	Integer
Default Value:	10
Valid Values:	Any integer greater than 0

The Pgm_Source_Admit_High option specifies the maximum number of multicast messages held on the admit queue before RTgms stops queuing additional messages. The admit queue is a queue of multicast messages waiting to be transmitted by RTgms using PGM. The messages in the queue might have been generated by RTgms or in response to a NAK.

If RTgms is stopped from sending data over PGM, SmartSockets buffering is used until RTgms can send messages over PGM again. When the number of messages waiting to be sent is greater than the value you specified for Pgm_Source_Admit_High, RTgms is flow controlled until the number of waiting messages drops below the value you specified for Pgm_Source_Admit_Low.

If you set a value too high for Pgm_Source_Admit_High, it can waste memory in RTgms. Too small a value can waste CPU time by causing RTgms to be stopped too often.

Pgm_Source_Admit_Low

Used for: RTgms processes only
Type: Integer
Default Value: 5
Valid Values: Any integer greater than 0

The Pgm_Source_Admit_Low option specifies the number of multicast messages left on the admit queue that triggers RTgms to be restarted. The RTgms had been stopped because the admit queue reached the value you set in Pgm_Source_Admit_High.

If you set a value too low for Pgm_Source_Admit_Low, the RTgms might not have enough time to prepare more data before the messages in the current admit queue are completely sent. This can cause uneven message delivery rates. Too high a value for Pgm_Source_Admit_Low can waste CPU time by causing RTgms to be started too often.

Pgm_Source_Group_Ttl

Used for: RTgms processes only
Type: Integer
Default Value: 1
Valid Values: Any integer between 1 and 255, inclusive

The Pgm_Source_Group_Ttl option specifies the initial time to live (TTL) for data sent by RTgms. This is one less than the maximum number of multicast routers that data sent by RTgms can travel through before reaching an RTclient.

The time to live (TTL) is automatically decremented every time the data passes through a router. The data is discarded by the router if, after it decrements the TTL, the TTL is 0. This allows data stuck in a loop between routers to eventually be eliminated. Too small a value for Pgm_Source_Group_Ttl can cause the data to be discarded too soon, before it ever reaches an RTclient. Too large a value for Pgm_Source_Group_Ttl can increase the amount of congestion caused by routing loops.

The default value of 1 only works if data does not pass through any routers to reach all RTclients. Consider the number of routers data must pass through to reach RTclients, including the number of routers to reach other networks containing RTclients, when setting the value for Pgm_Source_Group_Ttl.

Pgm_Source_Max_Trans_Rate

Used for:	RTgms processes only
Type:	Integer
Default Value:	4000000
Valid Values:	Any integer greater than 0

The Pgm_Source_Max_Trans_Rate option specifies the maximum rate, in bits a second (bps), at which RTgms attempts to send multicast messages. This value includes repair data sent in response to NAKs, in addition to data being sent for the first time. If RTgms attempts to send faster than this rate, data is queued on the admit queue up to the value you set for Pgm_Source_Admit_High.

Setting the value for Pgm_Source_Max_Trans_Rate improperly causes inefficient network use. If the value is smaller than the rate messages can be delivered by the network between RTgms and the RTclients, you are setting an artificial limit on the rate at which RTgms sends messages. This means the multicast messages are sent more slowly than they need to be. However, if you set the value higher than the rate messages can be delivered by the network between RTgms and the RTclients, this can cause excessive data loss in the network. More NAKs are generated, and more of the available network bandwidth is used for repair data, reducing the amount available to the original messages.

You can dynamically set the value for Pgm_Source_Max_Trans_Rate on a connection basis, using an ADMIN_SET message. For more information, see the information on the T_MT_GRP_ADMIN_SET_PGM_OPTIONS message.

Pgm_Source_Min_Trans_Rate

Used for:	RTgms processes only
Type:	Integer
Default Value:	56000
Valid Values:	Any integer greater than 0

The Pgm_Source_Min_Trans_Rate option specifies the minimum rate, in bits a second, at which RTgms can attempt to send multicast messages. This option is only useful when there is more than one RTgms running on the network.

Pgm_Source_Pgmcc

Used for:	RTgms processes only
Type:	Boolean
Default Value:	TRUE
Valid Values:	TRUE OR FALSE

The Pgm_Source_Pgmcc option specifies whether congestion control is on or off. The default, TRUE, turns congestion control on for RTgms processes sending multicast messages. This option must be set to FALSE if the RTgms process or any of the receiving RTclients are below SmartSockets Version 6.2. For more information about congestion control, see Bandwidth Management on page 647.

Pgm_Source_Pgmcc_Acker_Selection_Constant

Used for:	RTgms processes only
Type:	Integer
Default Value:	75
Valid Values:	Any integer from 1 to 100, inclusive

The `Pgm_Source_Pgmcc_Acker_Selection_Constant` option specifies the selection criteria for designating a new ACKer, based on throughput. If a receiving client's throughput is too low, its throughput is compared to the throughput of the existing ACKer, creating a percentage. That percentage is compared to the value you specified for `Pgm_Source_Pgmcc_Acker_Selection_Constant`. If the percentage is lower than the value you specified, that receiving client becomes the new ACKer. For example, the default value is 75. If a receiving client has throughput that is 73 percent of the current ACKer's throughput, that client becomes the new ACKer.

Pgm_Source_Pgmcc_Init_Acker

Used for:	RTgms processes only
Type:	String
Default Value:	0.0.0.0
Valid Values:	Any valid IP address

The `Pgm_Source_Pgmcc_Init_Acker` option specifies the IP address of the receiving RTclient that initially plays the role of the ACKer, until a new ACKer is chosen. Using the default value of 0.0.0.0 allows any receiving RTclient to be the initial ACKer, depending on which RTclient is the first to NAK and assuming it is allowed to be an ACKer. RTclients with the `Pgm_Receive_Pgmcc` option set to `FALSE` are not allowed to be ACKers.

Pgm_Udp_Encapsulation

Used for:	RTclient and RTgms processes
Type:	Integer
Default Value:	TRUE
Valid Values:	TRUE or FALSE

The Pgm_Udp_Encapsulation option specifies whether multicast messages are encapsulated directly in IP or if they are encapsulated in UDP before IP encapsulation. `TRUE` specifies that the messages are to be encapsulated in UDP. `FALSE` specifies IP encapsulation.

UDP encapsulation makes each multicast message 12 bytes longer, but the RTgms and RTclients do not need to run as `root` on UNIX systems. RTgms and RTclients in a multicast group must have the same value for the Pgm_Udp_Encapsulation option. Messages that are UDP encapsulated do not benefit from the Cisco Systems PGM Router Assist feature.

If Pgm_Udp_Encapsulation is set to 1 to use UDP encapsulation, no PGM subscribers can be run on the machine where RTgms is running. This is because NAKs from PGM might not be correctly processed by the UDP protocol when RTgms and a PGM subscriber are running on the same machine. This problem does not occur when PGM is using raw IP sockets (Pgm_Udp_Encapsulation set to 0).

Starting and Stopping RTgms

RTgms is normally started and stopped using the `rtgms` command, which works very much like the `rtserver` command you use to start an RTserver, which is covered in Starting and Stopping RTserver on page 284. Unlike the RTserver, RTgms cannot be automatically started by an RTclient through the use of the start prefix.

The syntax for the `rtgms` command is:

```
rtgms arg_list
```

where *arg_list* is optional and consists of one or more arguments, separated by a space:

<code>-check</code>	starts the non-optimized version of RTgms.
<code>-command <i>filename</i></code>	specifies that RTgms uses the local startup command file named <i>filename</i> . This overrides the name of the default local startup command file, <code>rtgms.cm</code> , but does not change the search order described in RTgms Startup Command Files on page 652.
<code>-help</code>	displays a description of the <code>rtgms</code> command arguments.
<code>-install -demandstart -autostart</code>	installs the RTgms process as a Windows service. You can specify either <code>demandstart</code> or <code>autostart</code> as the startup mode of the service. If you install RTgms as an autostart Windows service, RTgms starts automatically only when the machine is rebooted. This option is supported only on Windows systems.
<code>-no_console</code>	does not display a Windows console associated with the detached process. This option is supported only on Windows systems.
<code>-no_daemon</code>	runs RTgms as a foreground process.
<code>-password <i>pword</i></code>	provides the password used by RTgms, along with a password, to connect to RTserver when RTserver has Basic Security enabled. Use with the <code>-username</code> argument. <i>pword</i> size is unlimited. To specify no password, use empty quotation marks (" ")

<code>-server_names <i>server_list</i></code>	provides a list of RTservers to connect to. This overrides the list provided in the <code>Server_Names</code> option. The list is also used when you specify <code>-stop_all</code> or <code>-stop_rtgms</code> . <i>server_list</i> is a list of logical connection names for RTservers.
<code>-stop_all</code>	stops all RTgms processes. If specified together with <code>-server_names</code> , the list of servers provided is used to connect to those RTservers and stop all RTgms processes connected to those RTservers.
<code>-stop_rtgms <i>unique_subject</i></code>	stops the RTgms process identified by that unique subject. The location of the RTservers to use when looking for the RTgms to stop is specified in <code>-server_names</code> .
<code>-uninstall</code>	removes the RTgms as a Windows service. This option is only supported on Windows systems.
<code>-username <i>name</i></code>	provides the username used by RTgms, along with a password, to connect to RTserver when RTserver has Basic Security enabled. Use with the <code>-password</code> argument. <i>name</i> size is restricted to 64 characters.
<code>-verbose</code>	informs RTgms to output additional information messages such as, group up, group down, new member join, and member leave.
<code>-version</code>	prints a string that displays the latest build information.

Starting RTgms on UNIX

To run an RTgms process on UNIX:

1. Change to the directory in which RTgms will run.

The first thing you must do to begin running RTgms is to change from the current working directory to the one that contains the RTgms command file, `rtgms.cm`. If no `rtgms.cm` file is needed, then RTgms can be started from any directory.

To change directories, use:

```
$ cd directory
```

2. Create or edit the RTgms startup command files, if needed. See RTgms Startup Command Files on page 652.
3. Invoke the RTgms executable by typing the `rtgms` command at the operating system prompt. For example:

```
$ rtgms -command rtgms1.cm -verbose
```

When the startup processing is complete, the operating system prompt displays again.

Starting RTgms as a Service on Windows

If you installed RTgms on a Windows system, the RTgms process can be configured as a Windows service by adding the `-install` option to the start command.

This example configures the RTgms process to use the autostart mode of a Windows service:

```
$ rtgms -install -autostart
```

Once configured as a service, the RTgms process is managed through the standard service control manager of Windows.

To remove the RTgms process as a Windows service, you can use:

```
$ rtgms -uninstall
```

If you use the `-install` or `-uninstall` options when starting an RTgms process on a non-Windows system, such as Sun Solaris, you receive an error message indicating an invalid command line argument, and the process does not start.

Stopping RTgms

To stop a single RTgms, specify the unique subject of that RTgms in your `-stop` command and specify location of the RTserver to which it is connected in the `-server_names` argument. For example:

```
$ rtgms -stop_rtgms /rtgms_1 -server_names tcp:_node:5102
```

To stop all RTgms processes, use:

```
$ rtgms -stop_all
```

Interrupting RTgms

An RTgms process running in the foreground can be interrupted at any time while it is processing messages. Enter a Ctrl-c in the active RTgms window. This message is displayed:

```
rtgms interrupted by SIGINT signal (CONTROL-C).
```

Sending a Message using Multicast

To send a message using multicast, simply modify the `Server_Names` option setting for the RTclients to connect to an RTgms. No other code changes are required. The connection is now enabled for multicast.

For information on creating a connection from an RTclient to RTgms, see [Creating a Connection to RTgms](#) on page 190.

RTgms Commands

There are no new commands for RTgms. These commands are supported with RTgms:

- `alias`
- `cd`
- `echo`
- `edit`
- `help`
- `helpopt`
- `quit`
- `setopt`
- `sh`
- `source`
- `unalias`
- `unsetopt`

Use these commands as you would for any RTclient. Because RTgms is a type of RTclient, the commands work the same way as they do for any RTclient. For information on using these commands, see Chapter 9, Command Reference.

Tailoring Your Multicast Deployment

Multicast deployment is often more difficult than unicast deployment, and there are many issues that you need to consider when deploying multicast. The following sections provide information regarding the key issues that you need to address when deploying multicast. They include tips and information on bandwidth sharing, how network devices forward multicast traffic, and how different network environments affect your multicast deployment.

How Multicast Deployment Compares with Unicast Deployment

Even though connectivity for multicast and unicast applications is very similar, multicast deployment is often more difficult than unicast deployment.

Both multicast and unicast rely on the network layer for connectivity:

- multicast connectivity—multicast applications such as SmartSockets Multicast require the network layer to provide a path for multicast data to flow from senders to receivers
- unicast connectivity—web browsers require the network layer to provide unicast connectivity to web servers

However, this is why it is often more difficult to deploy multicast than unicast:

- older networking equipment may not be designed to accommodate multicast deployment. Examples of this are switches that can only flood multicast and routers that lack modern multicast routing protocols.
- different equipment solves the same problems in different ways. For example, some switches use IGMP snooping while others use CGMP.
- multicast diagnostic tools are not readily available
- network administrators may have less experience with multicast deployment than with unicast deployment
- administrators may need to configure bandwidth sharing for multicast deployment. In contrast, unicast streams automatically share bandwidth equitably, so administrators play no role in configuring bandwidth sharing.

Bandwidth Sharing

The bandwidth sharing mechanism that you use for multicasting depends on which version of SmartSockets you use. SmartSockets Versions 6.2 and higher support congestion control. Earlier versions of SmartSockets require administrators to configure the amount of bandwidth that they expect the network to deliver.

Bandwidth sharing for multicast is not automatic as it is for unicast. In unicast, reliable unicast transports (for example, TCP) automatically share available network bandwidth among all sessions contending for it. Administrators play no role in this process—protocol stacks measure the round-trip time and packet loss rates and dynamically determine available bandwidth. Unicast assumes that all streams have equal priority and automatically divides bandwidth accordingly.

TIBCO SmartSockets Version 6.2 and Higher

TIBCO SmartSockets, Versions 6.2 and higher, provide support for congestion control, which dynamically determines bandwidth limits and maximizes throughput. For information on setting congestion control options, see [Bandwidth Management](#) on page 647.

TIBCO SmartSockets Versions Prior to Version 6.2

SmartSockets versions prior to Version 6.2 do not provide support for congestion control. Instead, SmartSockets relies on administrators to configure the amount of bandwidth that they expect the network to deliver. If administrators fail to configure the amount of bandwidth, congestion can cause packet loss and either erratic behavior or application failure.

To optimize throughput, administrators need to limit how fast SmartSockets Multicast sends data. If you exceed your network's bandwidth capacity, the congestion causes the network to perform below its maximum capacity. For example, if you ask SmartSockets Multicast to deliver 11 Mbps over a 10 Mbps network layer, you may only receive 5 or 7 Mbps. In addition, you will probably experience chaotic behavior based on the loss rates and other factors. However, if you ask SmartSockets Multicast to deliver 9 Mbps over a 10 Mbps network layer, it will.

Here are some tips for bandwidth sharing in an environment that uses a SmartSockets version prior to Version 6.2:

- If you experience throughput that is below your network's bandwidth capacity, it is worthwhile to try reducing the SmartSockets Multicast transmission rate. If the sub-optimal throughput is due to congestion, a slower transmission rate may actually increase throughput.
- Remember that SmartSockets Multicast measures bandwidth for a stream. Use caution in environments where multiple SmartSockets Multicast streams share a network. The administrator must limit the total number of streams so that the network can keep up with the aggregate bandwidth.
- Use caution in environments where SmartSockets Multicast streams share a network with bursty unicast traffic. Because unicast generally attempts to take all available bandwidth on the link, bursty unicast traffic can cause higher SmartSockets Multicast loss rates. If your application requires SmartSockets Multicast traffic to have priority, you should place a limit on unicast traffic to guarantee SmartSockets Multicast its allocated piece of the total available bandwidth.

Client Failovers in Multicast

Client failovers using the multicast protocol, PGM, as the alternate protocol do not work. Because multicast uses threads on the client side, threading must be initialized before PGM connects to RTgms. To initialize threading, set the `Server_Names` option to `pgm:_node:your_value` to cause PGM to initialize threads when it loads.

For example, if the `Server_Names` option is set to `tcp,pgm:_node:your_value`, after the first successful TCP connection, RTclient stops traversing the `Server_Names` list until the existing TCP connection is closed. When RTclient loses the connection to RTserver, RTclient attempts to reconnect using TCP. If it cannot reconnect, RTclient connects using PGM. The PGM link driver loads, and threads are initialized.

RTclient cannot initialize threads in the middle of an application. Initializing threads in the middle of an application can cause core dumps. Threads must be initialized at program startup.

There is one workaround to this problem: call `TipcInitThreads` at program startup.

How Network Devices Forward Multicast

How multicast packets are forwarded depends on the types of network devices you use:

- simple physical-layer devices

Physical-layer devices like hubs that do not inspect packets to determine if they are unicast, broadcast, or multicast forward multicast packets to all stations on the network exactly as forward all other packets.

- intermediate physical-layer devices

Physical-layer devices that inspect packets far enough to know if the physical-layer address is unicast, broadcast, or multicast forward or "flood" multicast and broadcast packets to all ports. In contrast, they forward unicast packets only to the port containing the destination physical-layer address. A "dumb" switch is an example of such a device.

- advanced physical-layer devices

Advanced physical-layer devices use knowledge of which physical layer addresses are members of which network-layer multicast groups to selectively forward multicast packets. These devices monitor network-layer IGMP packets to obtain group membership information. This is often called "IGMP snooping."

Cisco Systems switches prefer to use a Cisco Systems proprietary protocol called "CGMP" instead. This protocol is used between routers and switches. Like IGMP snooping, it gives the switches the information needed to selectively forward multicast on a per-port basis instead of flooding it. It is easier for switches to run CGMP because it requires no network-layer work on their part.

- network-layer devices

Network-layer devices such as routers can generally be configured to forward multicast between attached networks even though this is not generally the default configuration. Once enabled, routers monitor IGMP group membership requests from hosts and forward traffic to ports as necessary. See Example Cisco Systems Router Configuration on page 679 for instructions on how to enable multicast forwarding for Cisco Systems routers.

Routers are also responsible for forwarding multicast traffic to interested devices that are not directly connected to their ports. A router can be configured to forward multicast traffic between its own ports even if it does not forward traffic to other routers. You must configure a multicast routing protocol such as PIM, DVMRP, or MOSPF before routers will distribute multicast traffic to all interested devices within an intranet.

Multicast deployment often also involves ensuring that multicast streams go only where they are wanted. This is especially important when high-bandwidth streams are present on a network with some low-bandwidth links or where access must be controlled at the network layer for security reasons. Within a LAN, all Ethernet switches can direct unicast traffic only to ports where it is wanted. However, many Ethernet switches simply flood multicast packets to all ports. Therefore, it may be necessary to configure your network to block the flow of multicast data for bandwidth-sharing or security reasons.

Testing for Multicast Traffic Before Configuring Your Network

Even before you configure your network specifically for multicast, your network may already pass multicast traffic in some areas. Before configuring your network, you may want to test your existing network to determine where multicast data is already flowing, if it is flowing at all. See [Multicast Troubleshooting](#) on page 487 for advice on testing multicast connectivity.

Multicast connectivity that happens without explicit network configuration generally sends all multicast traffic to all users on a LAN. See [Bandwidth Sharing](#) on page 674 for the implications of this.

If multicast traffic can already flow in your network, it is generally most likely to flow between users in close proximity to one another. For example, within a multi-story office building, offices on the same floor are very likely to find multicast connectivity. Likelihood of multicast connectivity diminishes as you move farther away: the minority of adjacent floors might have multicast connectivity, it is unlikely for floors farther away to have multicast connectivity, it is very unlikely to find multicast connectivity between buildings in a campus or across a wide-area network, and it is rare to find multicast connectivity between sites connected via the Internet unless both sites are educational or research sites.

Multicast and GMD

There are some known issues with the way multicasting works with GMD messages. If GMD messages are being multicast, and a subscribing client does a warm disconnect, some of those GMD messages might be missed. The problem occurs because the client notifies the RTgms process that it is disconnecting, and then the RTgms process notifies the RTserver sending the multicast GMD messages. Potentially, the RTserver might have sent several GMD messages after the client warm disconnect, but before the RTserver received notice from the RTgms process to start buffering the GMD messages.

There are two workarounds to this problem:

- if you are using multicast for GMD messages, ensure that each RTclient has the `Server_Disconnect_Mode` option set to either `gmd_failure` or `gmd_success`. This prevents any warm disconnects.
- ensure that the number of subscribing clients for GMD messages is always lower than the value you set for the `Group_Threshold` option. This ensures that the GMD messages are unicast, not multicast, avoiding the issue.

Also note, because of the overhead required to guarantee message delivery, multicasting with GMD is slower than multicasting without GMD.

UDP Encapsulation of PGM

SmartSockets Multicast allows PGM packets to be UDP encapsulated instead of being encapsulated directly in IP. UDP encapsulation of PGM packets means that:

- packets are larger by the size of a UDP header
- root privilege is not required (on UNIX)
- PGM packets are invisible to routers

Multicast Deployment with Frame Relay Networks

Frame relay networks are usually a partial or full mesh of point-to-point connections between end points. Switches in frame relay networks are committed to delivering a given bandwidth or Committed Information Rate (CIR) of frame relay circuits between endpoints. This means that they are unable to generate more than one packet out for each packet in, even though this is required to implement broadcast or multicast.

Because frame relay switches cannot broadcast or multicast, Cisco Systems routers can simulate it by duplicating all broadcast and multicast packets when sending to an interface for each remote end point. By default, a single serial interface transmits up to 3 point-to-point copies of a broadcast or multicast packet if it has 3 virtual circuits running over it. Be sure to allocate bandwidth for each multicast stream required by your application.

Some satellite applications use frame relay encapsulation but do not use frame relay switches. In satellite broadcast applications, all end points can hear traffic for all virtual circuits. For these applications, a Cisco Systems IOS frame-relay multicast `dlci` command can send a single copy of multicast packets to a DLCI shared by all end points. Be sure to use it on all serial ports that share the DLCI.

On terrestrial networks where packets must be copied for DLCI, Cisco Systems limits the broadcast bandwidth. The default limit of 36 broadcast packets a second equates to about 400 Kbps. Use the IOS frame-relay broadcast-queue command to increase the limit if needed. See the Cisco Systems tip page on the Frame Relay Broadcast Queue for more information.

Example Cisco Systems Router Configuration

Here is a sample configuration fragment for a Cisco Systems router that forwards multicast traffic between Ethernet interfaces with the PGM Router Assist function enabled on both interfaces.

```
ip multicast-routing
!
interface Ethernet 0
  ip pim sparse-dense-mode
  ip pgm router
!
interface Ethernet 1
  ip pim sparse-dense-mode
  ip pgm router
```

This also configures automatic discovery of all other PIM routers on Ethernet 0 and 1. Multicast traffic is exchanged among all PIM routers.

See the Cisco Systems *Multicast Quick-Start Configuration Guide* for additional Cisco Systems multicast configuration examples.

Index

Symbols

* wildcard in subjects 160
 _next logical connection name 196
 _random logical connection name 196
 _time subject 204

A

absolute subject names 159
 accepting connections 103, 105
 access control list
 See also ACL files
 accessing
 arrays in message fields 59
 messages 44
 acknowledging messages with GMD 338
 acknowledgment of delivery 312
 ACL files 275
 acl.cfg file 279
 address part of logical connection name 102, 194
 addresses 194
 sbrk 483
 admin group 279
 ADMIN_SET message
 RTgms 650
 RTserver 495
 ADMIN_SET message type 495
 alias command 591
 appending fields to message data 50
 architecture connection property 82
 array fields 59
 arrays as message fields
 accessing 59
 arrival timestamp message property 4
 ASCII character format 124
 Auth_Data_File option 514

authentication
 Proxy_Password option 553
 Proxy_Username option 553
 Authorize_Publish option 514
 auto flush size
 getting existing settings 76
 setting the value 76
 auto flush size connection property 73, 76, 120, 122, 209
 Auto Scroll, monitoring 452
 automatically connecting to RTserver 200

B

background processes 288
 backup files
 name extensions 515
 backup processes 429, 430
 Backup_Name option 515
 bandwidth management 647
 basic security 276
 big-endian integer layout 123
 binary buffers
 serializing messages into 66
 block mode
 and timeout properties 75
 blocking 152
 non-blocking 152
 block mode connection property 74, 77
 BOOLEAN_DATA message type 40
 buffer sizes
 Client_Max_Buffer option 519
 Group_Max_Buffer option 536

- buffering
 - infinite 76
 - messages
 - incoming 119
 - outgoing 122
 - specifying buffer size 559
- buffers
 - socket 87
 - write 120

C

- callback functions
 - TipcCbConnProcessGmdFailure 342, 357
 - TipcCbConnProcessKeepAlive 152
 - TipcCbConnProcessKeepAliveCall 151
 - TipcCbSrvError 201, 238
 - TipcCbSrvProcessControl 207
 - TipcCbSrvProcessGmdFailure 357
 - TipcConnDefaultCbCreate 108
 - TipcConnDefaultCbLookup 108
 - TipcConnErrorCbCreate 110
 - TipcConnErrorCbLookup 110
 - TipcConnProcessCbCreate 107
 - TipcConnProcessCbLookup 107
 - TipcConnQueueCbCreate 109
 - TipcConnQueueCbLookup 110
 - TipcConnReadCbCreate 108
 - TipcConnReadCbLookup 109
 - TipcConnWriteCbCreate 109
 - TipcConnWriteCbLookup 109
 - TipcSrv*CbCreate 205
 - TipcSrv*CbLookup 205
 - TipcSrvCreateCbCreate 200
 - TipcSrvCreateCbLookup 200
 - TipcSrvDestroyCbCreate 200, 238
 - TipcSrvDestroyCbLookup 200
 - TipcSrvSubjectCbCreate 205
 - TipcSrvSubjectCbDestroyAll 243
 - TipcSrvSubjectDefaultCbCreate 205, 242
 - TipcSrvTraverseCbCreate 188
 - TipcSrvTraverseCbLookup 188

- callbacks
 - connection 166, 205
 - default 111, 116
 - discussion 84
 - error 112, 238
 - write timeout 77
 - improper use of 479
 - internal 202
 - process 111, 341
 - GMD_FAILURE 342
 - queue 112
 - discussion 116
 - read 111
 - discussion 116
 - RTclient-specific 237
 - server create 198, 238
 - server destroy 202, 238
 - server names traverse 239
 - subject 205, 241
 - types 114
 - used to process polling results 388
 - used to process watch results 411
 - with warm connection 235
 - write 112, 120
- capturing messages 64
- case sensitivity xxv
 - on UNIX and Windows xxv
- catalog
 - resource
 - specifying name 515
- Catalog_File option 515
- Catalog_Flags option 516
- cd command 593
- changing
 - logging categories for message types 214
- changing directories 593
- character formats
 - ASCII 124
 - EBCDIC 124
- check mode
 - connection event 256
 - socket event 260
- checking
 - type of current field 58
- circular trace files 579

- Cisco routers
 - troubleshooting 490
- clear output command 452
- client connection 103
- Client failovers
 - in multicast 675
- Client_Burst_Interval option 516
- Client_Connect_Timeout option 517
- Client_Drain_Subjects option 517
- Client_Drain_Timeout option 518
- Client_Keep_Alive_Timeout option 519
- Client_Max_Buffer option 519
- Client_Max_Tokens option 520
- Client_Read_Timeout option 520
- Client_Reconnect_Timeout option 521
 - warm RTclients 346
- Client_Threads
 - options 522, 561
- Client_Token_Rate option 523
- client-server model 103
- cloning a message 59
- cloud of RTServers 300
- command files
 - mcast.cm 657
 - mcastopts.cm 657
 - RTclient startup 189
- command line, monitoring 452
- Command_Feedback option 523
- command-line arguments
 - debugging RTserver 486
- commands
 - alias 591
 - allowing in CONTROL messages 533
 - cd 593
 - clear output 452
 - connect 189, 595
 - create 597
 - credentials 599
 - disconnect 600
 - echo 602
 - edit 603
 - evaluate 604
 - exit 592
 - for debugging RTclient 483
 - for RTacl 589
 - for RTclients 585
 - for RTgms 672
 - for RTmon 587
 - for RTserver 584
 - groups 605
 - help 606
 - helpopt 607
 - history 608
 - load 609
 - ls 591
 - permissions 610
 - poll 611
 - processing from a file 623
 - pwd 591
 - quit 615
 - run 616
 - send 617
 - setnopt 620
 - setopt 619
 - sh 622
 - source 623
 - stats 625
 - subscribe 626
 - unalias 628
 - unsetopt 629
 - unsubscribe 630
 - unwatch 631
 - users 633
 - watch 634
 - with multiple connections 248
- comments in text message files 64
- communication between processes 72
- communication functions 174
- compression
 - by message type 272
 - Compression_Stats option 525
 - connection level 273
 - setting with LCNs 273
 - single message 272
- compression library
 - Compression_Name option 525
 - ZLIB 271
- compression message property 5
- Compression option 524
- compression property 28

- Compression_Args option 524
- Compression_Name option 525
- Compression_Stats option 525
- configuration files
 - acl.cfg file 279
 - groups.cfg file 278
 - users.cfg file 277
- configuring RTserver 290
- congestion control 647
 - enabling 665
- Conn_Max_Restarts option 526
- Conn_Names option 527
 - creating connections 290
 - multi-homed machines 527
- connect command 189, 595
- connect modifier 291
- connect prefix
 - connect_all 291
 - connect_all_stop 291
 - connect_one 291
- connect_all connect prefix 291
- connect_all_stop connect prefix 291
- CONNECT_CALL message type 351
- connect_one connect prefix 291
- connection callbacks 166, 205
- connection event
 - API summary 257
 - check mode 256
 - description 256
- connection level compression 273
 - Compression option 524
 - Compression_Args option 524
 - Compression_Name option 525
- connections
 - accepting 103, 105
 - advanced uses 141
 - advantages over sockets 88
 - assigning a cost
 - RTserver 292
 - callbacks 107, 205
 - error 201, 238
 - process 348, 357
 - write 336
 - checking for failures 151
 - client 103
 - compared to RTserver 169
 - compressing messages 273
 - controlling bandwidth 305
 - created by RTserver 290
 - creating 103, 150
 - creating a client 104
 - creating a server 104
 - debugging 479
 - defined 69
 - destroying 106
 - errors 110
 - example 89
 - failures 150
 - fault tolerance 149
 - handling network failures 149
 - infinite buffering 76
 - keep alive 151
 - limiting in RTserver 290
 - limiting server-to-server 547
 - logical names 101, 290, 307
 - address part 102, 194
 - connect prefix 291
 - extended 273
 - node part 101, 194
 - protocol part 101, 192
 - start prefix 195
 - losing due to busy process 575
 - maintain keep alive traffic 519
 - maximum limit 548
 - message queue 118
 - mixing with select function 143
 - mixing with sockets 143
 - mixing with Xt Intrinsics 141
 - multiple RTservers 248
 - file-based GMD 353
 - names used by clients 527
 - number of attempts to find a process 573
 - peer information
 - architecture property 82
 - node property 82
 - process ID property 82
 - unique subject property 83
 - user property 83
 - peer-to-peer 158, 315
 - peer-to-peer link 105

- priority queue 115
- properties 71
 - auto flush size 73, 76, 120, 122, 209
 - block mode 74, 77
 - callbacks 84
 - default callbacks 108
 - delivery timeout 11, 31, 79, 321, 335
 - error callbacks 110
 - GMD area 80, 316, 352
 - keep alive timeout 78
 - message queue 74
 - process callback 107
 - process mutex 81
 - queue callbacks 109
 - read buffer 73
 - read callbacks 108
 - read mutex 81
 - read timeout 77
 - socket 72
 - timeout 74
 - write buffer 73, 76
 - write callbacks 109
 - write mutex 81
 - write timeout 77
- reading data from 115
- reconnect delay 566
- restarting
 - number of times to try 526
- RTclient to RTserver 166, 189
- server 103, 290
- specifying automatic 559
- status 236
- threads 139
- to hosts with multiple IP addresses 102
- to RTservers with multiple IP addresses 197
- wait time for checking 565
- waiting for operations to complete 74
- warm 201, 202, 307, 346
- Xt Intrinsics 72
- constructing a message 44, 49
- CONTROL message type 40
 - security 279
- controlling network bandwidth 305

- convenience functions
 - TipcConnMainLoop 117
 - TipcConnMsgSearchType 56, 118
 - TipcConnMsgWrite 121
 - TipcMsgRead 54
 - TipcMsgWrite 51
- conventions used in this manual xxiii
- conversion of formats 123
- correlation ID message property 6
- CPU usage
 - stats command 625
- create command 597
- creating
 - client connection 104
 - connections 103
 - GMD area 334
 - message files 65
 - named options 620
 - RTserver connection 189
 - server connection 104
- creating a connection 595
- creating an alias 591
- creating message types 597
- credentials
 - See also security
- credentials command 599
- current field 52
 - accessing 52
 - changing 52
 - checking type of 58
 - setting 113
- current projects, monitoring 453
- customer support xxvi

D

- data
 - reading from connection 115
- data message property 7
- data parameter 113
- data standard message types 40
- DCE 297

- debug file
 - automatic removal 484
- debugging 282
 - connections 479
 - improper use of callbacks 479
 - limiting trace file size 579
 - messages 478
 - overview 477
 - permissions 282
 - RTclient commands for 483
 - RTclient not receiving data 481
 - RTclients 481
 - RTserver 484
 - VMS files created 485
 - RTserver options 486
 - specifying a security trace file format 556
 - specifying a trace file 578
 - specifying a trace file format 579
 - specifying amount of trace information 580
 - specifying Verbose option 581
 - unwanted messages 479
- default callback 116
- default callback connection property 108
- default IP address
 - udp_broadcast 197
- Default_Connect_Prefix option 528
- Default_Msg_Priority option 529, 597
- Default_Protocols option 529
 - TipcSrvCreate 190
- Default_Subject_Prefix option 159, 530
- defaults
 - for new message types 597
- definitions
 - acknowledgment 313
 - connection 69
 - current field 52
 - defunct subject 164
 - fault tolerance 149
 - GDI 446
 - keep alive 150
 - message 1
 - message type 2
 - project 156
 - remote procedure call 246
 - socket 86
 - subject 158
 - unwanted message 479
 - warm connection 235
- deleting
 - files from a GMD area 333
 - messages 343
- deleting aliases 628
- deleting subjects 630
- delivery mode property 30, 318, 336, 345
- delivery modes
 - best effort 9
 - GMD
 - all 9
 - some 9
 - ordered 9
- delivery timeout 597
 - getting existing settings 79
 - setting the value 79
- delivery timeout connection property 11, 31, 79, 321, 335
- delivery timeout failures 342
- delivery timeout message property 321, 335
- delivery timeouts
 - disabling 31, 79
- deriving new information 426
- designated ACKer
 - disabling 661
 - initial timeout 661
 - specifying initially 666
 - throughput criteria 666
- destination message property 12
- destroying
 - connection to RTserver 202
 - connections 106
 - message files 66
 - messages 44, 56
- destroying a connection 600
- detached processes 288
- diagnosing problems 477
- direct RTclients 298
- directories
 - changing 593
- directory services 428
- disconnect command 600

- DISCONNECT message
 - on destroy 202
- DISCONNECT message type 351
- dispatcher
 - API summary 250
 - description 249
 - example 264
 - in detached thread 249
 - listening on multiple connections 249, 251, 252
 - overview 248
- displaying
 - changes in monitoring information 634
 - command feedback 523
 - command information 606
 - current subject subscriptions 626
 - monitoring information 611
 - option information 607
 - previous commands 608
 - text 602
 - value of a named option 620
 - value of an option 619
 - verbose 581
- distributing messages 617
- domain naming service (DNS) 297
- duplicate messages 340
- dynamic message routing 296, 297

E

- EBCDIC character format 124
- echo command 602
- edit command 603
- Editor option
 - details 532
 - with edit command 603
- Enable_Control_Msgs option 533
 - processing control messages 207
- Enable_Stop_Msgs option 534
- ENUM_DATA message type 41
- error callback connection property 110
- errors
 - nonrecoverable 351
- Ethernet protocol 85
- evaluate command 604
- event
 - check mode 256, 260
 - connection event
 - APIs 257
 - description 256
 - description 254
 - message event
 - APIs 259
 - description 258
 - example 264
 - socket event
 - APIs 261
 - description 260
 - timer event
 - APIs 262
 - description 262
 - example 264
 - user event
 - APIs 263
 - description 263
- event flags 141
- events
 - multiple connections 248
- examples
 - compiling, linking, and running 47, 98, 136, 183, 233, 405, 423
 - compiling, linking, and running GMD 329
 - connecting to an RTserver 596
 - connections 89
 - disconnecting from RTserver 601
 - hot backup (Guardian program) 431
 - discussion 442
 - load balancing 220
 - message construction 44
 - polling 400
 - RTclient advanced features 227
 - RTserver and RTclient message sending 175
 - subscribing to subjects 627
 - watching 418
 - working with GMD 322
- executing shell commands 622
- exit command 592
- exiting
 - an RT process 615

- expiration message property 13
- extended LCNs
 - configuring compression 273
- extension 168
- extension data, RTclient 360, 362, 366, 382, 390
- exterior messages 301

F

- Failovers
 - client failovers in multicast 675

- failures
 - delivery timeout 342
 - GMD related 341
- fault tolerance 149, 307, 429
 - definition 149
 - hardware-based 149
 - software-based 149

- field types 7
 - binary 7
 - bool 8
 - bool_array 8
 - byte 8
 - char 7
 - for message values 8
 - int2 7
 - int2_array 7
 - int4 7, 8
 - int4_array 7
 - int8 7
 - int8_array 7
 - mismatches 478
 - msg 8
 - msg_array 8
 - pointer 61
 - real16 8
 - real16_array 8
 - real4 7
 - real4_array 7
 - real8 8
 - real8_array 8
 - str 7
 - str_array 7

- timestamp 8
- timestamp_array 8
- unknown values 63
- utf8 8
- utf8_array 8
- verbose 33
- xml 8
- fields 7
 - accessing 53
 - appending 50
 - array 59
 - containing messages 60
 - GMD_FAILURE message 341
 - number of 53
 - pointer 61
 - pointer-sized 54
 - repetitive group of 33
 - type 7
 - type mismatches 478
 - unknown values 63
 - value 8

- File command, monitoring 450
- file descriptors 105, 141, 143, 148

- file names
 - specifying xxv
- file-based credentials
 - credentials command 599
 - with basic security 282

- filenames
 - extensions for backup 515

- files
 - acl.cfg 279
 - created for debug 484
 - defining extensions for backup filenames 515
 - format of catalog strings 516
 - format of security trace file 556
 - format of trace file 579
 - GMD area on disk 539
 - groups.cfg 278
 - include 48
 - incoming data message log 541
 - incoming internal message log 542
 - incoming message log 540, 542
 - incoming multicast message log 541
 - incoming RTserver message log 543

- incoming status message log 543
- mcast.cm 657
- mcastopts.cm 657
- message 64
- outgoing data message log 544
- outgoing internal message log 545
- outgoing message log 544, 545
- outgoing multicast message log 544
- outgoing RTserver message log 545
- outgoing status message log 546
- reading commands from 623
- resource catalog 515
- rtgms.cm 652
- rtmon.cm 499
- rtserver.cm 498
- socket 199
- Trace_File option 578
- Trace_File_Size option 579
- users.cfg 277
- finding an RTserver 567
- finding RTservers 290, 293
- floating-point format 88
- flooding algorithm 302
- flow control
 - specifying end 663
 - specifying start 662
- for multicast 194
- formats
 - floating-point 88
 - for real numbers 554
 - for time 578
 - integer 88, 105, 116
 - big-endian 123
 - little-endian 123
 - real number 105, 116
 - DEC D 123
 - DEC F 123
 - DEC G 123
 - IEEE 123
 - strings 116
- functions
 - callback 113
 - TipcCbConnProcessGmdFailure 342
 - TipcCbConnProcessKeepAlive 152
 - TipcCbConnProcessKeepAliveCall 151
 - TipcConnErrorCbLookup 110
 - TipcConnQueueCbCreate 109
 - callbacks
 - callback-type-specific data argument 114
 - case-sensitivity xxv
 - communication 174
 - message 173
 - message type 172
 - monitoring
 - TipcMon* functions 143
 - poll 143
 - select 143, 147
 - system call 119, 122
 - TipcCbConnProcessGmdFailure 357
 - TipcCbSrvError 201, 238
 - TipcCbSrvProcessControl 207
 - TipcCbSrvProcessGmdFailure 357
 - TipcConn* 122, 158, 166, 167, 169, 186, 205, 207, 208
 - TipcConnAccept 74, 82, 83, 105, 106, 144, 146, 150, 333, 339
 - TipcConnCheck 76, 106, 143, 150, 151, 152, 342
 - TipcConnCreateClient 82, 83, 333, 339
 - TipcConnCreateClientProtocol 341
 - TipcConnDefaultCbCreate 108
 - TipcConnDefaultCbLookup 108
 - TipcConnDestroy 106, 341
 - TipcConnErrorCbCreate 110
 - TipcConnFlush 73, 74, 75, 76, 120, 122, 143, 339, 341
 - TipcConnGetArch 82
 - TipcConnGetBlockMode 74
 - TipcConnGetGmdMaxSize 335
 - TipcConnGetGmdNumPending 317, 338
 - TipcConnGetNode 82
 - TipcConnGetNumQueued 74
 - TipcConnGetPid 82
 - TipcConnGetSocket 72, 141, 143
 - TipcConnGetUniqueSubject 83
 - TipcConnGetUser 83
 - TipcConnGmdFileCreate 334, 336, 339
 - TipcConnGmdFileDelete 333, 339

- TipcConnGmdMsgDelete 317, 342
- TipcConnGmdMsgResend 342
- TipcConnGmdResend 317, 333, 339, 352
- TipcConnKeepAlive 151, 152
- TipcConnMainLoop 117, 141, 142, 144, 334, 338, 339
- TipcConnMsgInsert 57, 73, 74, 116
- TipcConnMsgNext 56, 74, 115, 116, 117, 119, 122, 142, 144, 147
- TipcConnMsgProcess 107, 108, 113, 115, 116, 117, 341
- TipcConnMsgSearch 56, 74, 116, 117, 118, 119
- TipcConnMsgSearchType 56, 118
- TipcConnMsgSend 73, 76, 120, 122, 143, 147, 317, 318, 334, 336, 339
- TipcConnMsgSendRpc 56, 147, 246
- TipcConnMsgWrite 121, 122
- TipcConnProcessCbCreate 107
- TipcConnProcessCbLookup 107
- TipcConnQueueCbLookup 110
- TipcConnRead 73, 74, 76, 115, 118, 119, 123, 124, 317, 334, 338, 342
- TipcConnReadCbCreate 108
- TipcConnReadCbLookup 109
- TipcConnSetBlockMode 74, 106
- TipcConnSetGmdMaxSize 80, 318, 335
- TipcConnSetSocket 72
- TipcConnWriteCbCreate 109
- TipcConnWriteCbLookup 109
- TipcDispatcherCreate 250
- TipcDispatcherCreateDetached 250
- TipcDispatcherDestroy 250
- TipcDispatcherDispatch 250
- TipcDispatcherMainLoop 250
- TipcDispatcherSrvAdd 250
- TipcDispatcherSrvRemove 250
- TipcEventCreateConn 257
- TipcEventCreateMsg 259
- TipcEventCreateMsgType 259
- TipcEventCreateSocket 261
- TipcEventCreateTimer 262
- TipcEventDestroy 257, 259
- TipcEventGetCheckMode 257, 261
- TipcEventGetConn 257
- TipcEventGetData 263
- TipcEventGetDispatcher 257, 259
- TipcEventGetInterval 262
- TipcEventGetSocket 261
- TipcEventGetType 257, 259
- TipcEventSetInterval 262
- TipcGetGmdDir 333
- TipcInitCommands 585
- TipcMon* 169
- TipcMonClientBufferPoll 382
- TipcMonClientBufferSetWatch 408
- TipcMonClientCbPoll 382
- TipcMonClientCpuPoll 382
- TipcMonClientExtPoll 382
- TipcMonClientGeneralPoll 382
- TipcMonClientInfoPoll 382
- TipcMonClientMsgRecvSetWatch 408
- TipcMonClientMsgSendSetWatch 408
- TipcMonClientMsgTrafficPoll 383
- TipcMonClientMsgTypeExPoll 383
- TipcMonClientNamesNumPoll 383
- TipcMonClientNamesPoll 383
- TipcMonClientNamesSetWatch 409
- TipcMonClientOptionPoll 383
- TipcMonClientSubjectExPoll 383
- TipcMonClientSubscribeNumPoll 383
- TipcMonClientSubscribePoll 383
- TipcMonClientSubscribeSetWatch 409
- TipcMonClientTimePoll 383, 385
- TipcMonClientTimeSetWatch 409
- TipcMonClientVersionPoll 383
- TipcMonProjectNamesPoll 382, 611
- TipcMonServerBufferPoll 383
- TipcMonServerConnPoll 383
- TipcMonServerConnSetWatch 409
- TipcMonServerCpuPoll 384
- TipcMonServerGeneralPoll 384
- TipcMonServerMsgTrafficExPoll 384
- TipcMonServerNamesPoll 384
- TipcMonServerNamesSetWatch 409
- TipcMonServerOptionPoll 384
- TipcMonServerRoutePoll 384
- TipcMonServerStartTimePoll 384
- TipcMonServerTimePoll 384
- TipcMonServerVersionPoll 384
- TipcMonSubjectNamesPoll 384
- TipcMonSubjectNamesSetWatch 409

TipcMonSubjectSubscribePoll 384, 385
 TipcMonSubjectSubscribeSetWatch 409
 TipcMsg* 120
 TipcMsgAck 317, 338
 TipcMsgAppendMsg 60
 TipcMsgClone 20, 56, 59
 TipcMsgCreate 49, 56
 TipcMsgDestroy 20, 56, 57, 115, 117, 338
 TipcMsgFileCreate 65
 TipcMsgFileDestroy 66
 TipcMsgFileRead 56, 65
 TipcMsgFileWrite 64, 65
 TipcMsgGetCompression 5
 TipcMsgGetDeliveryMode 10
 TipcMsgGetDeliveryTimeout 11
 TipcMsgGetDest 12, 21
 TipcMsgGetExpiration 13
 TipcMsgGetHeaderStrEncode 14
 TipcMsgGetLbMode 15
 TipcMsgGetNumFields 17, 53
 TipcMsgGetPriority 18
 TipcMsgGetReadOnly 19
 TipcMsgGetRefCount 20
 TipcMsgGetSender 22
 TipcMsgGetSenderTimestamp 23
 TipcMsgGetSeqNum 4, 6, 16, 24
 TipcMsgGetType 25
 TipcMsgGetUserProp 26
 TipcMsgIncrRefCount 20, 56
 TipcMsgNextMsg 60
 TipcMsgNextType 58
 TipcMsgRead 54, 60
 TipcMsgSetCompression 5
 TipcMsgSetCurrent 52
 TipcMsgSetDelivery 10
 TipcMsgSetDeliveryMode 336
 TipcMsgSetDeliveryTimeout 11
 TipcMsgSetDest 12, 21, 208
 TipcMsgSetExpiration 13
 TipcMsgSetHeaderStrEncode 14
 TipcMsgSetLbMode 15
 TipcMsgSetNumFields 17, 57
 TipcMsgSetPriority 18
 TipcMsgSetSender 22
 TipcMsgSetSenderTimestamp 23
 TipcMsgSetType 25
 TipcMsgSetUserProp 26
 TipcMsgWrite 51, 59
 TipcMtCreate 43
 TipcMtGetCompression 29
 TipcMtGetDeliveryMode 30
 TipcMtGetDeliveryTimeout 31
 TipcMtGetGrammar 32
 TipcMtGetHeaderStrEncode 34
 TipcMtGetLbMode 35
 TipcMtGetName 36
 TipcMtGetNum 37
 TipcMtGetPriority 38
 TipcMtGetUserProp 39
 TipcMtSetCompression 28
 TipcMtSetDeliveryMode 30, 336
 TipcMtSetDeliveryTimeout 31
 TipcMtSetHeaderStrEncode 34
 TipcMtSetPriority 38
 TipcMtSetPriorityUnknown 38
 TipcMtSetUserProp 39
 TipcSrv* 167, 169, 186, 200, 205, 207, 208, 235, 237
 TipcSrv*CbCreate 205
 TipcSrv*CbLookup 205
 TipcSrvCreate 189, 190, 198, 200, 201, 207, 235, 236, 346, 351
 TipcSrvCreateCbCreate 200
 TipcSrvCreateCbLookup 200
 TipcSrvDestroy 200, 202, 236, 238, 346, 351
 TipcSrvDestroyCbCreate 200, 238
 TipcSrvDestroyCbLookup 200
 TipcSrvGetConnStatus 200, 236
 TipcSrvGmdFileCreate 352
 TipcSrvGmdFileDelete 200, 352
 TipcSrvGmdMsgServerDelete 347, 357
 TipcSrvGmdMsgStatus 348, 349, 358
 TipcSrvGmdResend 200
 TipcSrvIsRunning 200
 TipcSrvLogAddMt 200, 210, 214
 TipcSrvLogRemoveMt 200, 210, 214
 TipcSrvMsgInsert 57
 TipcSrvMsgNext 56
 TipcSrvMsgSearch 56
 TipcSrvMsgSearchType 56, 246, 349
 TipcSrvMsgSend 208, 209, 246

- TipcSrvMsgSendRpc 56
- TipcSrvMsgWrite 208, 209
- TipcSrvMsgWriteVa 208
- TipcSrvSetAutoFlushSize 209
- TipcSrvStdSubjectSetSubscribe 204, 577
- TipcSrvStop 200
- TipcSrvSubject* 143
- TipcSrvSubjectCbCreate 205
- TipcSrvSubjectCbDestroyAll 243
- TipcSrvSubjectDefaultCbCreate 205, 242
- TipcSrvSubjectGetSubscribe 203
- TipcSrvSubjectGmdInit 354
- TipcSrvSubjectLbInit 216
- TipcSrvSubjectSetSubscribe 203
- TutCommandParseFile 189
- TutCommandParseStr 207, 585
- TutCommandParseTypedStr 585
- TutGetSocketDir 199
- TutOptionSetUnknown 214
- utility 174
 - TutSocketCheck 143
- Xt Intrinsic
 - TipcConnGetXtSource 141
 - XtAppAddInput 141, 143
 - XtAppAddTimeOut 142
 - XtAppAddWorkProc 142
 - XtAppMainLoop 141, 143
 - XtAppPending 142
 - XtAppProcessEvent 142
 - XtRemoveInput 141

G

GDI

- defining prompt 552
- getting help
 - help command 606
- getting values
 - for auto flush size 76
 - for delivery timeout 79
 - for keep alive timeout 78
 - for read timeout 77
 - for write timeout 78

GMD

- acknowledging delivery 347
- acknowledgment messages 313
- asynchronous operation 317
- automatic acknowledgement 538
- automatic acknowledgement policy 539
- Client_Reconnect_Timeout option 521
- combined with monitoring 356
- components
 - sequence number 315
- delivery modes 345
- delivery timeout property 31
- disconnect actions 563
- duplicate sequence numbers 337
- error codes 347
- file-based 80, 316
 - configuration 331
 - multiple RTserversconsiderations 353
 - reverting to memory 333
 - specifying directory 564
- keeping accounting information 534
- limitations 343
- load balancing 219
- memory-only 80, 316
- message types 347
- no subscribers 582
- pending messages 337
- polling for status 347
- priority with GMD 343
- processing failure messages 357
- receiver processing 317
- restrictions 343
- sender processing 316
- setting the value for GMD area 318
- specifying file or memory 540
- specifying GMD area 539
- terminating for a specific message 357
- time to acknowledge 562
- waiting for completion 338
- warm RTclients 346
- with multicast 678
- with multiple connections 248

- GMD area 120
 - creating 334
 - deleting files from 333
 - highest sequence number 352
 - maximum size 335
- GMD area connection property 80, 316
- GMD_ACK message type 41, 320, 347, 354
- GMD_DELETE message type 41, 347, 355
- GMD_FAILURE message fields 341
- GMD_FAILURE message type 41, 321, 347, 348, 355, 357
- GMD_FAILURE messages 341
- gmd_failure setting
 - Server_Disconnect_Mode option 563
- GMD_INIT_CALL message type 41
- GMD_INIT_RESULT message type 41
- GMD_NACK message type 348, 355
- Gmd_Publish_Timeout option 534
 - using for RTserver 354
- GMD_STATUS_CALL message type 41, 348, 355
- GMD_STATUS_RESULT message type 41, 349, 355
- gmd_success setting
 - Server_Disconnect_Mode option 563
- GMD-related failures 341
- grammar message type property 32
- graph
 - undirected 299
- group 299
- Group_Burst_Interval option 535
- Group_Max_Buffer option 536
- Group_Max_Tokens option 536
- Group_Names option 537
- Group_Threshold option 656
- Group_Token_Rate option 538
- groups
 - admin group 279
 - basic security 278, 279
 - multicast addresses 194
 - names of RTservers 537
 - setting permissions 279
- groups command 605
- GRP_ADMIN_SET_OUTBOUND_RATE_PARAMS
 - message type 650
- GRP_ADMIN_SET_PGM_OPTIONS message
 - type 651

- guaranteed message delivery
 - see also GMD 309

H

- hardware failures 149
- header string encode message property 14
- help command 606
- Help command, monitoring 451
- helpopt command 607
- heterogeneous environment 123
- heterogeneous networks 85
- hierarchical subject names 159
- hierarchical subject namespace 159
- history command 608
- hosts
 - multi-homed 102
- hot backup processes 208, 429, 430

I

- identification strings 427
 - Monitor_Ident option 549
- identifiers
 - case sensitivity xxv
- identifying
 - specifying Unique_Subject 581
- IEEE floating-point format 123
- include files 48, 100, 186
- indirect RTclients 298
- information
 - deriving 426
- integer formats 88, 105, 116
 - big-endian 123
 - incompatible 123
 - little-endian 123
- interface
 - defining prompt 552
- interior messages 301
- internal callbacks 202
- internal standard message types 40

IP address

- machines with multiple 102
- multi-homed machines 527
- RTservers with multiple 197

Ip_Gmd_Auto_Ack option 538

Ip_Gmd_Auto_Ack_Policy option 539

Ip_Gmd_Directory option 539

GMD 334

Ip_Gmd_Type option 540

J**JMS support**

- message types 41

JMS_BYTES message type 41

JMS_MAP message type 41

JMS_OBJECT message type 41

JMS_STREAM message type 42

JMS_TEXT message type 42

K**keep alive 77**

- Client_Keep_Alive_Timeout option 519
- definition 150
- Server_Keep_Alive_Timeout option 565
- Server_Read_Timeout option 571
- TCP/IP 150
- timeout 150

keep alive timeout

- disabling 78
- getting existing settings 78
- setting the value 78

keep alive timeout connection property 78

KEEP_ALIVE_CALL message type 78, 151

KEEP_ALIVE_RESULT message type 78, 151

keywords

- _any 527

L

latency 648

LCN

- See also logical connection names

libraries

- compression library 271

little-endian integer layout 123

load balancing 215

GMD 219

overriding 217

load balancing message property 15

load command 609

load_balancing_off 217

local protocol 85, 150, 163, 195

- connect problem 199

localhost

- specifying connection to RTgms 191

location

- of mcast.cm file 657

Log_In_Client option 540

Log_In_Data option 541

Log_In_Group option 541

Log_In_Internal option 542

Log_In_Msgs option 542

Log_In_Server option 543

Log_In_Status option 543

Log_Out_Client option 544

Log_Out_Data option 544

Log_Out_Group option 544

Log_Out_Internal option 545

Log_Out_Msgs option 545

Log_Out_Server option 545

Log_Out_Status option 546

logging

- messages 210
- messages by RTserver 295

- logical connection names 101, 290, 307
 - address part 102
 - configuring compression 273
 - connect prefix 291
 - extended 273
 - finding other RTservers 293
 - modifiers 291
 - next 196
 - node part 101
 - protocol part 101, 192
 - randomizing 196
 - start prefix 195
- looking up message types 42
- loss rate
 - percentage of gain 662
- loss, recovering from 648
- lost date
 - multicast NAKs 660
- ls command 591

M

- Max_Client_Conns option 546
- Max_Server_Accept_Conns option 547
- Max_Server_Connect_Conns option 547
- Max_Server_Conns option 548
- maximum size of GMD area 335
- mcast_cm_file option 658
- memory usage
 - stats command 625
- message
 - compression
 - interior 301
 - properties
 - arrival timestamp 4
 - correlation ID 6
 - message ID 16
 - routing 296, 301
- message compression 271
 - See also compression
- message data fields See fields
- message event
 - API summary 259
 - description 258
 - example 264
- message files 40, 64, 108, 109, 210, 295
 - creating 65
 - destroying 66
 - grammar 32
 - reading 65
 - text 64
 - comments in 64
 - using 65
- message functions 173
- message ID message property 16
- message logging
 - starting 213
 - stopping 214
- message queue 109, 118
 - adding message to 74
 - getting message from 74
 - priority 74
 - searching for specific message 74
- message queue connection property 74
- message type
 - creating 597
- message type functions 172
- message types 27, 384
 - ADMIN_SET_OUTBOUND_RATE_PARAMS 495
 - call and result numbers for RPC 246
 - CALL and RESULT pairs 385
 - categories
 - data 212
 - internal 212
 - changing logging categories 214
 - CONNECT_CALL 351
 - CONTROL
 - processing 207
 - security 279
 - defined 2
 - DISCONNECT 351
 - GMD 320, 347
 - GMD_ACK 320, 347, 354
 - GMD_DELETE 347, 355
 - GMD_FAILURE 321, 347, 348, 355, 357
 - GMD_NACK 348, 355

GMD_STATUS_CALL 348, 355
 GMD_STATUS_RESULT 349, 355
 GRP_ADMIN_SET_OUTBOUND_RATE_PARAMS 650
 GRP_ADMIN_SET_PGM_OPTIONS 651
 KEEP_ALIVE_CALL 78, 151
 KEEP_ALIVE_RESULT 78, 151
 logging categories 210
 looking up 42
 MON_* 356
 MON_*_POLL_CALL 362, 367
 MON_*_POLL_RESULT 362, 367
 MON_*_SET_WATCH 362, 367
 MON_*_STATUS 362, 367
 MON_CLIENT_BUFFER_POLL_CALL 367, 389
 MON_CLIENT_BUFFER_POLL_RESULT 367, 382, 389
 MON_CLIENT_BUFFER_SET_WATCH 367, 413
 MON_CLIENT_BUFFER_STATUS 367, 408, 413
 MON_CLIENT_CB_POLL_CALL 367, 389
 MON_CLIENT_CB_POLL_RESULT 368, 382
 MON_CLIENT_CONGESTION_SET_WATCH 368, 414
 MON_CLIENT_CONGESTION_STATUS 368, 414
 MON_CLIENT_CPU_POLL_CALL 368, 390
 MON_CLIENT_CPU_POLL_RESULT 368, 382, 390
 MON_CLIENT_EXT_POLL_CALL 368
 MON_CLIENT_EXT_POLL_RESULT 368, 382
 MON_CLIENT_GENERAL_POLL_CALL 368, 390
 MON_CLIENT_GENERAL_POLL_RESULT 369, 382, 390
 MON_CLIENT_INFO_POLL_CALL 369, 391
 MON_CLIENT_INFO_POLL_RESULT 370, 382, 391
 MON_CLIENT_MSG_RECV_SET_WATCH 370, 414
 MON_CLIENT_MSG_RECV_STATUS 370, 408, 414
 MON_CLIENT_MSG_SEND_SET_WATCH 370, 414
 MON_CLIENT_MSG_SEND_STATUS 371, 414
 MON_CLIENT_MSG_TRAFFIC_POLL_CALL 371, 391
 MON_CLIENT_MSG_TRAFFIC_POLL_RESULT 371, 383, 391
 MON_CLIENT_MSG_TYPE_EX_POLL_CALL 371, 391
 MON_CLIENT_MSG_TYPE_EX_POLL_RESULT 372, 383, 391
 MON_CLIENT_MSG_TYPE_POLL_CALL 372
 MON_CLIENT_MSG_TYPE_POLL_RESULT 373
 MON_CLIENT_NAMES_NUM_POLL_CALL 373, 392
 MON_CLIENT_NAMES_NUM_POLL_RESULT 383, 392
 MON_CLIENT_NAMES_POLL_CALL 373, 392
 MON_CLIENT_NAMES_POLL_RESULT 373, 383, 392, 427
 MON_CLIENT_NAMES_SET_WATCH 373, 415
 MON_CLIENT_NAMES_STATUS 374, 409, 415, 427
 MON_CLIENT_OPTION_POLL_CALL 374, 392
 MON_CLIENT_OPTION_POLL_RESULT 374, 383, 392
 MON_CLIENT_SUBJECT_EX_POLL_CALL 374, 393
 MON_CLIENT_SUBJECT_EX_POLL_RESULT 374, 383
 MON_CLIENT_SUBJECT_POLL_CALL 374
 MON_CLIENT_SUBJECT_POLL_RESULT 375, 393
 MON_CLIENT_SUBSCRIBE_NUM_POLL_CALL 375, 393
 MON_CLIENT_SUBSCRIBE_NUM_POLL_RESULT 375, 383, 393
 MON_CLIENT_SUBSCRIBE_POLL_CALL 375, 393
 MON_CLIENT_SUBSCRIBE_POLL_RESULT 375, 383, 393
 MON_CLIENT_SUBSCRIBE_SET_WATCH 375, 415
 MON_CLIENT_SUBSCRIBE_STATUS 375, 409, 415
 MON_CLIENT_TIME_POLL_CALL 375, 385, 394
 MON_CLIENT_TIME_POLL_RESULT 375, 383, 385, 394
 MON_CLIENT_TIME_SET_WATCH 376, 410, 415
 MON_CLIENT_TIME_STATUS 376, 409, 415
 MON_CLIENT_VERSION_POLL_CALL 376, 394
 MON_CLIENT_VERSION_POLL_RESULT 376, 383, 394, 398
 MON_PROJECT_NAMES_POLL_CALL 376, 394

- MON_PROJECT_NAMES_POLL_RESULT 376, 382, 394
- MON_PROJECT_NAMES_SET_WATCH 376, 416
- MON_PROJECT_NAMES_STATUS 376, 408, 416
- MON_SERVER_BUFFER_POLL_CALL 376, 395
- MON_SERVER_BUFFER_POLL_RESULT 376, 383, 395
- MON_SERVER_CONGESTION_SET_WATCH 377, 416
- MON_SERVER_CONGESTION_STATUS 377, 416
- MON_SERVER_CONN_POLL_CALL 377, 395
- MON_SERVER_CONN_POLL_RESULT 377, 383, 395
- MON_SERVER_CONN_SET_WATCH 377, 416
- MON_SERVER_CONN_STATUS 377, 409, 416
- MON_SERVER_CPU_POLL_CALL 377, 395
- MON_SERVER_CPU_POLL_RESULT 377, 384, 395
- MON_SERVER_GENERAL_POLL_CALL 378, 396
- MON_SERVER_GENERAL_POLL_RESULT 378, 384, 396
- MON_SERVER_MAX_CLIENT_LICENSES_SET_WATCH 378, 417
- MON_SERVER_MAX_CLIENT_LICENSES_STATUS 378, 417
- MON_SERVER_MSG_TRAFFIC_EX_POLL_CALL 378, 396
- MON_SERVER_MSG_TRAFFIC_EX_POLL_RESULT 379, 384
- MON_SERVER_MSG_TRAFFIC_POLL_CALL 379, 380
- MON_SERVER_MSG_TRAFFIC_POLL_RESULT 379, 396
- MON_SERVER_NAMES_POLL_CALL 379, 396
- MON_SERVER_NAMES_POLL_RESULT 379, 384, 396, 427
- MON_SERVER_NAMES_SET_WATCH 379, 417
- MON_SERVER_NAMES_STATUS 379, 409, 417, 427
- MON_SERVER_OPTION_POLL_CALL 379, 397
- MON_SERVER_OPTION_POLL_RESULT 380, 397
- MON_SERVER_ROUTE_POLL_CALL 397
- MON_SERVER_ROUTE_POLL_RESULT 380, 384, 397
- MON_SERVER_START_TIME_POLL_CALL 380, 397
- MON_SERVER_START_TIME_POLL_RESULT 380, 384, 397
- MON_SERVER_TIME_POLL_CALL 380, 398
- MON_SERVER_TIME_POLL_RESULT 380, 384, 398
- MON_SERVER_VERSION_POLL_CALL 380, 398
- MON_SERVER_VERSION_POLL_RESULT 380, 384
- MON_SUBJECT_NAMES_POLL_CALL 380, 398
- MON_SUBJECT_NAMES_POLL_RESULT 381, 384, 398
- MON_SUBJECT_NAMES_SET_WATCH 381, 417
- MON_SUBJECT_NAMES_STATUS 381, 409, 417
- MON_SUBJECT_SUBSCRIBE_POLL_CALL 381, 385, 399
- MON_SUBJECT_SUBSCRIBE_POLL_RESULT 381, 384, 385, 399
- MON_SUBJECT_SUBSCRIBE_SET_WATCH 381, 410, 418
- MON_SUBJECT_SUBSCRIBE_STATUS 381, 409, 418
- monitoring 364, 367
 - list of 367
 - priority 363
- polling 389
- printing information about 478
- processed locally by RTserver 293
- properties
 - compression 28
 - delivery mode 30
 - grammar 32
 - name 36
 - number 37, 147
 - priority 38
 - user-defined 39
- restricted 279
- standard 27, 40, 214
 - BOOLEAN_DATA 40
 - CONTROL 40
 - data 40
 - ENUM_DATA 41
 - GMD_ACK 41
 - GMD_DELETE 41
 - GMD_FAILURE 41
 - GMD_INIT_CALL 41

- GMD_INIT_RESULT 41
- GMD_STATUS_CALL 41
- GMD_STATUS_RESULT 41
- internal 40
- JMS_BYTES 41
- JMS_MAP 41
- JMS_OBJECT 41
- JMS_STREAM 42
- JMS_TEXT 42
- NUMERIC_DATA 42
- status 40
- STRING_DATA 42
- user-defined 43, 295
- watching 413
- messages
 - ability to send 567
 - accessing 44
 - accessing in nonpriority order 117
 - acknowledging with GMD 338
 - acknowledgment of delivery 312
 - as fields in other messages 60
 - buffering
 - incoming 119
 - outgoing 122
 - callbacks 84
 - capturing 64
 - case sensitivity xxv
 - cloning 59
 - compressing
 - constructing 44, 49
 - CONTROL
 - setting allowable commands 533
 - converting 123
 - creating 49
 - current field
 - setting 113
 - debugging 478
 - defined 1
 - deleting 343
 - destroying 44, 56
 - duplicate 116, 340
 - exterior 301
 - fields
 - number of 53
 - GMD completion 338
 - GMD delivery timeout 562
 - GMD_FAILURE 341
 - GMD_FAILURE fields 341
 - heterogeneous environment 123
 - incoming
 - buffering 143
 - managed by a dispatcher 249
 - storing 73, 74
 - logging 210
 - logging by RTserver 295
 - logging categories for RTserver 295
 - logging incoming 540, 542
 - logging incoming data 541
 - logging incoming internal 542
 - logging incoming multicast 541
 - logging incoming RTserver messages 543
 - logging incoming status 543
 - logging outgoing 544, 545
 - logging outgoing data 544
 - logging outgoing internal 545
 - logging outgoing multicast 544
 - logging outgoing RTserver messages 545
 - logging outgoing status 546
 - logging to file 66
 - logging to file for debug 478
 - message queue
 - adding message to 74
 - getting message from 74
 - searching for specific message 74
 - modifying 19
 - outgoing
 - buffering 143
 - delivery mode 30
 - delivery timeout 31
 - header string encode 34
 - load balancing mode 35
 - priority 38
 - storing 73
 - user-defined property 39
 - write callbacks 109
 - pointer fields 61
 - printing information about 478
 - processing 113, 115, 207
 - using a convenience function 117
 - processing with run 616

- properties
 - compression 5
 - data 7
 - delivery mode for GMD 318
 - delivery mode GMD example 336
 - delivery mode with GMD 345
 - delivery modes
 - message property 9
 - delivery timeout 65
 - delivery timeout for GMD 321
 - delivery timeout GMD example 335
 - destination 12, 158
 - expiration 13
 - header string encode 14
 - load balancing 15
 - num fields 17
 - priority 74
 - read only 19
 - reference count 20, 65
 - reply to 21
 - sender 22, 158
 - sender timestamp 23
 - sequence number 24, 315, 352
 - type 25
 - user-defined 26
- receiving 207
- receiving with GMD 337
- remote procedure call 147
- rereceiving 340
- resending 342
- resending old GMD 339
- reusing 57
- routing by RTserver to RTclient 293
- routing example 164
- searching for a specific message type 118
- searching for specific 117
- send command 617
- sending 120, 123, 208
 - in heterogeneous environment 123
- sending with GMD 336
- setting non-data properties 49
- setting wait period for next message 116
- tracing lost 482
- types in log file 211
- unwanted 479
 - receiving 479
- value
 - field type 8
- waiting for GMD completion 338
- modifying a message 19
- MON_* message type 356
- MON_*_POLL_CALL message types 362, 367
- MON_*_POLL_RESULT message types 362, 367
- MON_*_SET_WATCH message types 362, 367
- MON_*_STATUS message types 362, 367
- MON_CLIENT_BUFFER_POLL_CALL message
 - type 367, 389
- MON_CLIENT_BUFFER_POLL_RESULT message
 - type 367, 382, 389
- MON_CLIENT_BUFFER_SET_WATCH message
 - type 367, 413
- MON_CLIENT_BUFFER_STATUS message type 367, 408, 413
- MON_CLIENT_CB_POLL_CALL message type 367, 389
- MON_CLIENT_CB_POLL_RESULT message
 - type 368, 382
- MON_CLIENT_CONGESTION_SET_WATCH message type 368, 414
- MON_CLIENT_CONGESTION_STATUS message
 - type 368, 414
- MON_CLIENT_CPU_POLL_CALL message
 - type 368, 390
- MON_CLIENT_CPU_POLL_RESULT message
 - type 368, 382, 390
- MON_CLIENT_EXT_POLL_CALL message type 368
- MON_CLIENT_EXT_POLL_RESULT message
 - type 368, 382
- MON_CLIENT_GENERAL_POLL_CALL message
 - type 368, 390
- MON_CLIENT_GENERAL_POLL_RESULT message
 - type 369, 382, 390
- MON_CLIENT_INFO_POLL_CALL message
 - type 369, 391
- MON_CLIENT_INFO_POLL_RESULT message
 - type 370, 382, 391
- MON_CLIENT_MSG_RECV_SET_WATCH message
 - type 370, 414

- MON_CLIENT_MSG_RECV_STATUS message type 370, 408, 414
- MON_CLIENT_MSG_SEND_SET_WATCH message type 370, 414
- MON_CLIENT_MSG_SEND_STATUS message type 371, 414
- MON_CLIENT_MSG_TRAFFIC_POLL_CALL message type 371, 391
- MON_CLIENT_MSG_TRAFFIC_POLL_RESULT message type 371, 383, 391
- MON_CLIENT_MSG_TYPE_EX_POLL_CALL message type 371, 391
- MON_CLIENT_MSG_TYPE_EX_POLL_RESULT message type 372, 383, 391
- MON_CLIENT_MSG_TYPE_POLL_CALL message type 372
- MON_CLIENT_MSG_TYPE_POLL_RESULT message type 373
- MON_CLIENT_NAMES_NUM_POLL_CALL message type 373, 392
- MON_CLIENT_NAMES_NUM_POLL_RESULT message type 383, 392
- message types
 - MON_CLIENT_NAMES_NUM_POLL_RESULT 373
- MON_CLIENT_NAMES_POLL_CALL message type 373, 392
- MON_CLIENT_NAMES_POLL_RESULT message type 373, 383, 392, 427
- MON_CLIENT_NAMES_SET_WATCH message type 373, 415
- MON_CLIENT_NAMES_STATUS message type 374, 409, 415, 427
- MON_CLIENT_OPTION_POLL_CALL message type 374, 392
- MON_CLIENT_OPTION_POLL_RESULT message type 374, 383, 392
- MON_CLIENT_SUBJECT_EX_POLL_CALL message type 374, 393
- MON_CLIENT_SUBJECT_EX_POLL_RESULT message type 374, 383
- MON_CLIENT_SUBJECT_POLL_CALL message type 374
- MON_CLIENT_SUBJECT_POLL_RESULT message type 375, 393
- MON_CLIENT_SUBSCRIBE_NUM_POLL_CALL message type 375, 393
- MON_CLIENT_SUBSCRIBE_NUM_POLL_RESULT message type 375, 383, 393
- MON_CLIENT_SUBSCRIBE_POLL_CALL message type 375, 393
- MON_CLIENT_SUBSCRIBE_POLL_RESULT message type 375, 383, 393
- MON_CLIENT_SUBSCRIBE_SET_WATCH message type 375, 415
- MON_CLIENT_SUBSCRIBE_STATUS message type 375, 409, 415
- MON_CLIENT_TIME_POLL_CALL message type 375, 385, 394
- MON_CLIENT_TIME_POLL_RESULT message type 375, 383, 385, 394
- MON_CLIENT_TIME_SET_WATCH message type 376, 410, 415
- MON_CLIENT_TIME_STATUS message type 376, 409, 415
- MON_CLIENT_VERSION_POLL_CALL message type 376, 394
- MON_CLIENT_VERSION_POLL_RESULT message type 376, 383, 394, 398
- MON_PROJECT_NAMES_POLL_CALL message type 376, 394
- MON_PROJECT_NAMES_POLL_RESULT message type 376, 382, 394
- MON_PROJECT_NAMES_SET_WATCH message type 376, 416
- MON_PROJECT_NAMES_STATUS message type 376, 408, 416
- MON_SERVER_BUFFER_POLL_CALL message type 376, 395
- MON_SERVER_BUFFER_POLL_RESULT message type 376, 383, 395
- MON_SERVER_CONGESTION_SET_WATCH message type 377, 416
- MON_SERVER_CONGESTION_STATUS message type 377, 416
- MON_SERVER_CONN_POLL_CALL message type 377, 395
- MON_SERVER_CONN_POLL_RESULT message type 377, 383, 395

- MON_SERVER_CONN_SET_WATCH message
 - type 377, 416
- MON_SERVER_CONN_STATUS message type 377, 409, 416
- MON_SERVER_CPU_POLL_CALL message
 - type 377, 395
- MON_SERVER_CPU_POLL_RESULT message
 - type 377, 384, 395
- MON_SERVER_GENERAL_POLL_CALL message
 - type 378, 396
- MON_SERVER_GENERAL_POLL_RESULT message
 - type 378, 384, 396
- MON_SERVER_MAX_CLIENT_LICENSES_SET_WATCH message type 378, 417
- MON_SERVER_MAX_CLIENT_LICENSES_STATUS message type 378, 417
- MON_SERVER_MSG_TRAFFIC_EX_POLL_CALL message type 378, 396
- MON_SERVER_MSG_TRAFFIC_EX_POLL_RESULT message type 379, 384
- MON_SERVER_MSG_TRAFFIC_POLL_CALL message type 379
- MON_SERVER_MSG_TRAFFIC_POLL_RESULT message type 379, 396
- MON_SERVER_NAMES_POLL_CALL message
 - type 379, 396
- MON_SERVER_NAMES_POLL_RESULT message
 - type 379, 384, 396, 427
- MON_SERVER_NAMES_SET_WATCH message
 - type 379, 417
- MON_SERVER_NAMES_STATUS message type 379, 409, 417, 427
- MON_SERVER_OPTION_POLL_CALL message
 - type 379, 397
- MON_SERVER_OPTION_POLL_RESULT 384
- MON_SERVER_OPTION_POLL_RESULT message
 - type 380, 384, 397
- MON_SERVER_ROUTE_POLL_CALL message
 - type 380, 397
- MON_SERVER_ROUTE_POLL_RESULT message
 - type 380, 384, 397
- MON_SERVER_START_TIME_POLL_CALL message
 - type 380, 397
- MON_SERVER_START_TIME_POLL_RESULT message type 380, 384, 397
- MON_SERVER_TIME_POLL_CALL message
 - type 380, 398
- MON_SERVER_TIME_POLL_RESULT message
 - type 380, 384, 398
- MON_SERVER_VERSION_POLL_CALL message
 - type 380, 398
- MON_SERVER_VERSION_POLL_RESULT message
 - type 380, 384
- MON_SUBJECT_NAMES_POLL_CALL message
 - type 380, 398
- MON_SUBJECT_NAMES_POLL_RESULT message
 - type 381, 384, 398
- MON_SUBJECT_NAMES_SET_WATCH message
 - type 381, 417
- MON_SUBJECT_NAMES_STATUS message type 381, 409, 417
- MON_SUBJECT_SUBSCRIBE_POLL_CALL message
 - type 381, 385, 399
- MON_SUBJECT_SUBSCRIBE_POLL_RESULT message type 381, 384, 385, 399
- MON_SUBJECT_SUBSCRIBE_SET_WATCH message
 - type 381, 410, 418
- MON_SUBJECT_SUBSCRIBE_STATUS message
 - type 381, 409, 418
- Monitor_Ident option 549
- Monitor_Scope option 550
 - using to monitor 365
- monitoring
 - advanced 426
 - blocking 364
 - by RTclients 307
 - combined with GMD 356
 - command bar 452
 - current state 452
 - extension data 360, 362, 366, 382, 390
 - Graphical Development Interface (GDI) 446
 - incoming messages 456
 - indentification strings 549
 - information available 360
 - initiating 361
 - items that do not exist 364
 - location of requested information 362

- managing command files 453
- menu bar 450
- message types 364, 367
 - priority 363
- multiple responses 364
- options 364
- poll command 611
- polling or watching 366
- RTclient by RTclient 168
- RTclient message traffic 470
- RTclients 454
- rtmon.cm file 448, 453
- RTserver by RTclient 166
- RTserver information 465
- RTserver response to request 361
- send a message 471
- sequential list mode 457
- specifying categories 550
- standard message types 362
- standard output 452
- stopping 474
- subjects 161, 203, 364
- time to return information 362
- TipcMon* functions 143
- view incoming messages 459
- view log file 458
- watch outgoing messages 460
- watch results 411
- watching 366
- watching.description 408
- Motif 141, 238
- Multi_Threated_Mode
 - options 551
- Multicast
 - client failovers 675
 - with GMD 678
- multicast
 - address field 194
 - enabling congestion control 665
 - flow control
 - specifying high water mark 662
 - specifying low water mark 663
 - Group_Names option 537
 - logging incoming messages 541
 - logging outgoing messages 544

- loss rate 662
- minimum transmission rate 665
- root access 667
- specifying data time to live 663
- specifying NAK time to live 660
- specifying port 659
- specifying threshold 656
- specifying transmission rate 664
- using localhost 191
- multi-homed hosts 102, 197
- multi-homed machines
 - listening to all IP addresses 527
- multiple connections 248
 - specifying GMD directory 564
 - using a dispatcher 249, 252
 - using commands with 248
 - without a dispatcher 251
- multiple RTserver connections 248
- multiple RTserver processes 298
- multiple RTservers 293, 296
- multithreading
 - specifying number of threads 569

N

- NAK 648
- name message type property 36
- named options
 - creating 620
 - setting, displaying 620
- naming services 428
- negative acknowledgement 648
- network bandwidth 298
- network chatter 300
- network failures 149, 166, 307, 313
 - automatic checking for 151
 - checking for 31, 77, 79
 - detecting 150
 - handling by connections 149
 - handling by sockets 87

- keep alive query 77
- undetected 152
- unreceived data 151
- unsent data 150
- network protocols 85, 312
- networks
 - bandwidth control 305
 - heterogeneous 85
- new message types 597
- node connection property 82
- node part of logical connection name 101, 194
- non-blocking operations 74
- num fields message property 17
- number message type property 37
- NUMERIC_DATA message type 42

O

- ODATA 648
- option
 - Client_Drain_Timeout 518
- options
 - associating with a name 620
 - Auth_Data_File 514
 - Authorize_Publish 514
 - Backup_Name 515
 - case sensitivity xxv
 - Catalog_File 515
 - Catalog_Flags 516
 - changing dynamically 247
 - Client_Burst_Interval 516
 - Client_Connect_Timeout 517
 - Client_Drain_Subjects 517
 - Client_Keep_Alive_Timeout 519
 - Client_Max_Buffer 519
 - Client_Max_Tokens 520
 - Client_Read_Timeout 520
 - Client_Reconnect_Timeout 521
 - warm RTclients 346
 - Client_Threads 522, 561
 - Client_Token_Rate 523
 - Command_Feedback 523
 - Compression 524
 - Compression_Args 524
 - Compression_Name 525
 - Compression_Stats 525
 - Conn_Max_Restarts 526
 - Conn_Names 527
 - Default_Connect_Prefix 528
 - Default_Msg_Priority 529
 - Default_Protocols 529
 - Default_Subject_Prefix 159, 530
 - Editor 532
 - Enable_Control_Msgs 533
 - processing control messages 207
 - Enable_Stop_Msgs 534
 - for debugging RTclient 482
 - for debugging RTserver 486
 - for multicast 657
 - Gmd_Publish_Timeout 534
 - using for RTserver 354
 - Group_Burst_Interval 535
 - Group_Max_Buffer 536
 - Group_Max_Tokens 536
 - Group_Names 537
 - Group_Threshold 656
 - Group_Token_Rate 538
 - Ipc_Gmd_Auto_Ack 538
 - Ipc_Gmd_Auto_Ack_Policy 539
 - Ipc_Gmd_Directory 539
 - GMD area 334
 - Ipc_Gmd_Type 540
 - Log_In_Client 540
 - Log_In_Data 541
 - Log_In_Group 541
 - Log_In_Internal 542
 - Log_In_Msgs 542
 - Log_In_Server 543
 - Log_In_Status 543
 - Log_Out_Client 544
 - Log_Out_Data 544
 - Log_Out_Group 544
 - Log_Out_Internal 545
 - Log_Out_Msgs 545
 - Log_Out_Server 545
 - Log_Out_Status 546
 - Max_Client_Conns 546
 - Max_Server_Accept_Conns 547

- Max_Server_Connect_Conns 547
- Max_Server_Conns 548
- mcast_cm_file 658
- Monitor_Ident 549
- Monitor_Scope 550
- monitoring 364
- Multi_Threaded_Mode 551
- named
 - multiple connections 248
 - setnopt command 620
- Pgm_Port 659
- Pgm_Receive_Nak_Ttl 660
- Pgm_Receive_Pgmcc 661
- Pgm_Receive_Pgmcc_Acker_Interval 661
- Pgm_Receive_Pgmcc_Loss_Constant 662
- Pgm_Source_Admit_High 662
- Pgm_Source_Admit_Low 663
- Pgm_Source_Group_Ttl 663
- Pgm_Source_Max_Trans_Rate 664
- Pgm_Source_Min_Trans_Rate 665
- Pgm_Source_Pgmcc 665
- Pgm_Source_Pgmcc_Acker_Selection_Constant 666
- 6
- Pgm_Source_Pgmcc_Init_Acker 666
- Pgm_Udp_Encapsulation 667
- Project 157, 552
- Prompt 552
- Proxy_Password 553
- Proxy_Username 553
- Real_Number_Format 554
- RTserver GMD-related 355
- Sd_Basic_Acl 554
- Sd_Basic_Acl_Timeout 555
- Sd_Basic_Admin_Msg_Types 555
- Sd_Basic_Trace_File 556
- Sd_Basic_Trace_Flags 556
- Sd_Basic_Trace_Level 557
- Sender_Get_Reply 557
- Server_Async_Subscribe 558
- Server_Auto_Connect 559
- Server_Auto_Flush_Size 559
- Server_Burst_Interval 560
- Server_Connect_Timeout 560
- Server_Delivery_Timeout 562
- Server_Disconnect_Mode 563
- configuring GMD 350
- warm RTclients 346
- Server_Gmd_Dir_Name 564
- Server_Keep_Alive_Timeout 565
- Server_Max_Reconnect_Delay 566
- Server_Max_Tokens 566
- Server_Msg_Send 567
- Server_Names 567
- Server_Num_Threads 569
- Server_Read_Timeout 571
- Server_Reconnect_Interval 572
- Server_Start_Delay 572
- Server_Start_Max_Tries 573
- Server_Start_Timeout 573
- Server_Threads 574
- Server-Token_Rate 574
- Server_Write_Timeout 575
- setopt command 619
- setting 189
- setting for RTgms 650
- setting RTserver 290
- Sm_Security_Driver 575
- Socket_Connect_Timeout 576
 - handling network failures 150
- ss.monitor_level 549
- Subjects 577
- summary
 - RTclient 501
 - RTgms 653
 - RTmon 509
 - RTserver 505
- Time_Format 578
- Trace_File 578
- Trace_File_Size 579
- Trace_Flags 579
- Trace_Level 580
- TutCommandParseStr 189
- Udp_Broadcast_Timeout 580
- Unique_Subject 162, 581
 - file-based GMD 331
- unsetopt command 629
- Verbose 581
- Zero_Recv_Gmd_Failure 582
- original data 648
- OSPF 297

- overriding
 - load balancing 217

P

- packets
 - messages into binary 66
- password
 - setting for basic security 282
- pathnames
 - for file-based GMD area 352
- peer-to-peer
 - connection 158, 315
 - link 105
 - model 103
- permissions 279
 - debugging with RTaCl 282
 - permissions command 610
 - using wildcards 281
- PGM
 - client failovers 675
- PGM options
 - setting for each connection 651
- Pgm_Port option 659
- Pgm_Receive_Nak_Ttl option 660
- Pgm_Receive_Pgmcc option 661
- Pgm_Receive_Pgmcc_Acker_Interval option 661
- Pgm_Receive_Pgmcc_Loss_Constant option 662
- Pgm_Source_Admit_High option 662
- Pgm_Source_Admit_Low option 663
- Pgm_Source_Group_Ttl option 663
- pgm_source_max_trans_rate 647
- Pgm_Source_Max_Trans_Rate option 664
- pgm_source_min_trans_rate 647
- Pgm_Source_Min_Trans_Rate option 665
- Pgm_Source_Pgmcc option 665
- Pgm_Source_Pgmcc_Acker_Selection_Constant option 666
- Pgm_Source_Pgmcc_Init_Acker option 666
- pgm_source_transmit_size_buffer 647
- Pgm_Udp_Encapsulation option 667
- pipes 88
- pointer fields 61

- poll command 611
- Poll command, monitoring 451
- poll function 143
- polling
 - for GMD status 347
 - GMD status of message 348
 - processing results 386
 - by searching 386
 - with callbacks 388
- polling message types 389
- ports
 - specifying for multicast 659
- printf
 - real number formats 554
- printing
 - number formats 554
- printing monitoring categories 412
- prioritizing throughput versus reliability 648
- priority
 - for new message types 597
 - message type property
 - default 38
 - using with GMD 343
- priority message type property 38
- priority queue 115
- process callback connection property 107
- process callbacks 341
- process ID connection property 82
- process mutex connection property 81
- processes
 - background 288
 - backup 429, 430
 - detached 288
 - hot backup 429, 430
 - identification string 427
- processing
 - CONTROL messages 207
 - GMD failure messages 357
 - messages 207
 - messages with run command 616
- Project Name command, monitoring 453
- Project option 157, 552
 - belonging to a project 191
- projects 191
 - definition 156

- Prompt option 552
- protocol part of logical connection name 101
- protocols 101, 192, 290
 - Ethernet 85
 - local 85, 150, 163, 195
 - connect problem 199
 - network 312
 - TCP/IP 85, 150
 - token ring 85
 - udp_broadcast 193
- proxy servers
 - providing password 553
 - providing userid 553
- Proxy_Password option 553
- Proxy_Username option 553
- pseudo field types 33
- publish-subscribe 158
 - with wildcards 159
- pwd command 591

Q

- queue callback 116
- queue callback connection property 109
- quit command 615

R

- randomizing server names 196
- rate control
 - network bandwidth 305
- rate control, throughput 647
- RDATA 648
- read buffer
 - size 73
- read buffer connection property 73
- read callback 116
- read callback connection property 108
- read mutex connection property 81
- read only message property 19
- read operations 152

- read timeout
 - getting existing settings 77
- read timeout connection property 77
- read timeouts
 - disabling 77
- real number formats 105, 116
 - DEC D 123
 - DEC F 123
 - DEC G 123
 - IEEE 123
 - incompatible 123
- Real_Number_Format option 554
- real-time database 300
- receiving
 - messages 207
- receiving messages with GMD 337
- reconnecting to RTserver 237, 293
 - automatically 201
- recovering from loss 648
- reference count message property 20
- reliability
 - running with hot backup 429
- reliability versus throughput, prioritizing 648
- remote procedure calls 117, 147, 246
 - blocking 147
 - keep alive 151
 - non-blocking 147, 246
- renaming commands 591
- repair data 648
- repetitive group of fields 33
- reply to message property 21
- rereceiving messages 340
- resending messages 342
- resending old GMD messages 339
- resource
 - catalog
 - specifying name 515
- restarting RTserver 307
- restricted message types 279
- reusing messages 57
- root access
 - RTgms 667
- routing
 - shortest path 296

- RPC 147
 - See also remote procedure calls
- RTacli 282
 - supported commands 589
- RTclient
 - callbacks 237
 - changing disconnect mode 351
 - connection to RTserver 166
 - debugging 482
 - description 166
 - direct 298
 - disconnecting from RTserver 350, 351
 - dispatcher 249
 - enabling ACKer for multicast 661
 - enabling message sends 567
 - events 249
 - executing commands 585
 - file-based GMD area 352
 - indirect 298
 - initially subscribing 577
 - load balancing 215
 - monitoring a project 307
 - monitoring extension data 360, 362, 366, 382, 390
 - not receiving data 481
 - Project option 552
 - receiving keep alives 520
 - reconnecting to RTserver 346, 350, 355
 - resending GMD messages 352, 354
 - running when RTserver down 237
 - specifying Unique_Subject option 581
 - start RTserver
 - as Windows service 195
 - timeout for data 571
 - user-defined 189
 - wait for broadcast 580
 - warm 346, 355
 - warm disconnect 350
- RTgms
 - as a Windows service 668
 - controlling bandwidth rate 305
 - flow control
 - specifying high water mark 662
 - specifying low water mark 663
 - GMD issues 678
 - running without root access 667
 - setting options 650
 - for a connection 651
 - specifying data time to live 663
 - specifying localhost 191
 - specifying port 659
 - specifying transmission rate size 664
 - specifying uni or multi-cast 656
 - starting 668
 - stopping 671
 - transmission rate 665
 - Windows service 670
- rtlink shell script 47, 98, 183
- RTmon
 - Auto Scroll command 452
 - command bar 452
 - command line 452
 - current projects 453
 - current state 452
 - defining prompt 552
 - executing commands 587
 - exiting 615
 - File command 450
 - Graphical Development Interface (GDI) 446
 - Help command 451
 - main menu bar 450
 - managing command files 453
 - Poll command 451
 - Project Name command 453
 - RTclient message traffic 470
 - rtmon.cm file 448, 453
 - RTserver information 465
 - Run command 450
 - send a message 471
 - sequential list mode 457
 - specifying Unique_Subject option 581
 - standard output 452
 - stopping 474
 - to debug 478
 - View command 450
 - view incoming messages 459
 - view log file 458
 - Watch command 450
 - watch incoming messages 456
 - watch outgoing messages 460
 - watch RTclients 454

- watching 408
 - when to use 366
- rtmon.cm file 448, 453
- RTserver
 - ADMIN_SET message 495
 - automatic connections 559
 - automatically connecting to 200
 - automatically reconnecting on error 201
 - broadcast waits 580
 - client keep alive 520
 - cloud 300
 - configuring 290
 - connecting to 291, 595
 - connection example 596
 - controlling bandwidth rate 305
 - creating connections 290
 - debugging options 486
 - defining prompt 552
 - defining RTserver groups 537
 - description 163
 - destroying connection to 202
 - disconnect example 601
 - disconnecting from 600
 - distributed database 300
 - enabling stop command 534
 - executing commands 584
 - exiting 615
 - finding 193
 - finding other RTservers 290, 293
 - GMD information 354
 - GMD-related options 355
 - group 299
 - how GMD works 354
 - information tables 163
 - interior messages 301
 - locally processed message types 293
 - maximum inbound server-to-server
 - connections 547
 - maximum outbound server connections 547
 - maximum RTclients 546
 - maximum server connections 548
 - message routing 354
 - multi-homed 197
 - multiple 163, 296, 298
 - multiple connections 248
 - multiple processes 293
 - multi-thread mode 569
 - number to restart connection 526
 - options
 - setting for a connection 495
 - reconnect interval 572
 - reconnecting to RTserver 293
 - resending GMD messages 350, 355
 - restarting 307
 - routing 296
 - server connections 290
 - setting options 290
 - specified in Server_Names 567
 - specifying GMD disconnect actions 563
 - specifying Unique_Subject option 581
 - starting 195, 288
 - starting on a remote node 199
 - stopping background process 289
 - terminating GMD for a message 355
 - timeout for data 571
 - tolerance for busy processes 575
 - udp_broadcast and multiple IP addresses 197
 - waiting for connect message 560
 - warm connection to 235, 346
 - with an interactive command interface 287
- RTserver and RTclient
 - function 154
- RTserver subscribe 299
- rtserver.cm files 288
- rtserver64 command 184, 199, 282, 284, 289
- RTservers
 - connecting to 248
 - multiple connections 248
- run command 616
- Run command, monitoring 450

S

- sbrk address 483
- scanning server names 196
- Sd_Basic_Acl option 554
- Sd_Basic_Acl_Timeout option 555
- Sd_Basic_Admin_Msg_Types option 555

- Sd_Basic_Trace_File option 556
- Sd_Basic_Trace_Flags option 556
- Sd_Basic_Trace_Level option 557
- sdbasic.cm 499
- searching for a specific message 117
- security 275, 282
 - acl.cfg file 279
 - basic security 276
 - configuration files 276
 - groups.cfg file 278, 279
 - permissions 279
 - Proxy_Password option 553
 - Proxy_Username option 553
 - RTacl 282
 - setting username and password 282
 - username and password based 276
 - users.cfg file 277
- select function 143, 147
- send command 617
- sender message property 22
- sender timestamp message property 23
- Sender_Get_Reply option 557
- sending
 - messages 120, 208
 - messages with GMD 336
- sequence number message property 24
- serializing messages 66
- server connections 103, 290
- server create callbacks 198, 238
- server destroy callbacks 202, 238
- server names traverse callbacks 239
- server process
 - number of allowed connections 105
- Server_Async_Subscribe option 558
- Server_Auto_Connect option 559
 - using 200
- Server_Auto_Flush_Size option 559
- Server_Burst_Interval option 560
- Server_Connect_Timeout option 560
- Server_Delivery_Timeout option 562
 - message types 597
- Server_Disconnect_Mode option 563
 - configuring GMD 350
 - TipcSrvCreate 190
 - warm RTclients 346
- Server_Disconnect_Mode options
 - and load balancing 219
- Server_Gmd_Dir_Name option 564
- Server_Keep_Alive_Timeout option 565
- Server_Max_Reconnect_Delay option 566
- Server_Max_Tokens option 566
- Server_Msg_Send option 567
 - creating backup processes 208
 - running a backup RTclient 430
 - using 208
- Server_Names option 567
 - finding other RTservers 293
 - number of times to try 573
 - reconnect interval 572
 - sleeping between traversals 572
 - specifying proxy servers 192
 - TipcSrvCreate 190
- Server_Num_Threads option 569
 - overriding value 286
- Server_Read_Timeout option 571
- Server_Reconnect_Interval option 572
- Server_Start_Delay option 572
 - TipcSrvCreate 190
- Server_Start_Max_Tries option 573
 - TipcSrvCreate 190
- Server_Start_Timeout option 573
 - TipcSrvCreate 190
- Server_Threads
 - options 574
- Server_Token_Rate option 574
- Server_Write_Timeout option 575
- setnopt command 620
- setopt command 619
- setting options
 - named
 - setnopt command 620
 - setopt command 619
 - unsetopt command 629
- setting properties
 - for auto flush size 76
 - for delivery timeout 79
 - for GMD area 318
 - for keep alive timeout 78
 - for write timeout 78
- sh command 622

- shared memory 88
- shell
 - defaults for operating systems 622
- shell commands
 - executing with sh 622
 - specifying xxv
- shortest-path routing 296, 301
- size
 - read buffer 73
 - write buffer 73
- sm_security_driver
- Sm_Security_Driver option 575
- SmartPGM 647
- SNMP monitoring 426
- socket connection property 72
- socket descriptors 148
- socket event
 - API summary 261
 - check mode 260
 - description 260
- socket file 199
- Socket_Connect_Timeout option 576
 - handling network failures 150
- sockets 85
 - buffers 87
 - compared to connections 88
 - datagram 86, 313
 - definition 86
 - handling network failures 87
 - how they work 87
 - loss of data 312
 - OpenVMS implementation 86
 - raw 86
 - stream 86, 313
 - wait 576
- software failures 149
- software redundancy 429
- source command 623
- specifying location of mcast file 658
- speed
 - setting transmission rate for multicast 664
- ss.monitor_level option 549
- standard message types 40, 214
- standard subjects 162, 204
- start prefix 195
 - start_always 195
 - start_never 195
 - start_on_demand 195
- start_always start prefix 195
- start_never start prefix 195
- start_on_demand start prefix 195
- starting
 - message logging 213
 - RTserver 195, 288
 - RTserver on a remote node 199
- startup command files
 - hot backup example 431
 - RTclient 498
 - RTgms 652
 - RTmon 499
 - RTserver 498
- stats command 625
- status message types 40
- stopping
 - background RTserver 289
 - message logging 214
- stopping RTserver
 - Enable_Stop_Msgs option 534
- string formats 116
 - incompatible 123
- STRING_DATA message type 42
- subject callbacks 205, 241
- subjects 160
 - * wildcard 160
 - _time 204
 - absolute names 159
 - definition 158
 - defunct 164
 - hierarchical names 159
 - hierarchical namespace 159
 - initial subscription 577
 - message event 258
 - monitoring 161, 203, 364
 - peer-to-peer connections 162
 - standard 162, 204
 - subscribe command 626
 - subscribing to 158, 203
 - unsubscribe command 630
 - user-defined 163

- wildcard examples 160
- wildcards 159
- Subjects option 577
- subscribe
 - RTserver 299
- subscribe command 626
- subscribing to a subject 203
- support, contacting xxvi
- system call functions 119, 122

T

- T_IO_CHECK_READ 256, 260
- T_IO_CHECK_WRITE 256, 260
- T_IPC_EVENT_CONN 257
- T_IPC_EVENT_MSG 259
- T_IPC_EVENT_MSG_TYPE 259
- T_IPC_EVENT_SOCKET 261
- T_IPC_EVENT_TIMER 262
- T_IPC_EVENT_USER 263
- T_IPC_LB_NONE 218
- T_IPC_LB_ROUND_ROBIN 218
- T_IPC_LB_SORTED 218
- T_IPC_LB_WEIGHTED 218
- TCP/IP protocol 85, 150
 - keepalive 150
- technical support xxvi
- text
 - displaying 602
- text editor
 - invoking 603
- text message files 64
- threads
 - with connections 81, 125, 139
 - with RTclient 226
- throughput rate control 647
- throughput versus reliability, prioritizing 648
- time
 - specifying your own format 578
- time converter
 - specifying in Time_Format option 578
- time resolution 147
- Time_Format option 578

- timer event
 - API summary 262
 - description 262
 - example 264
- TipcCbConnProcessGmdFailure function 342, 357
- TipcCbConnProcessKeepAlive function 152
- TipcCbConnProcessKeepAliveCall function 151
- TipcCbSrvError function 201, 238
- TipcCbSrvProcessControl function 207
- TipcCbSrvProcessGmdFailure function 357
- TipcConn* function 122, 158, 166, 167, 169, 186, 205, 207, 208
- TipcConnAccept function 74, 82, 83, 105, 106, 144, 146, 150, 333, 339
- TipcConnCheck function 76, 106, 143, 150, 151, 152, 342
- TipcConnCreateClient function 82, 83, 333, 339
- TipcConnCreateClientProtocol function 341
- TipcConnDefaultCbCreate function 108
- TipcConnDefaultCbLookup function 108
- TipcConnDestroy function 106, 341
- TipcConnErrorCbCreate function 110
- TipcConnErrorCbLookup function 110
- TipcConnFlush function 73, 74, 75, 76, 120, 122, 143, 339, 341
- TipcConnGetArch function 82
- TipcConnGetBlockMode function 74
- TipcConnGetGmdMaxSize function 335
- TipcConnGetGmdNumPending function 317, 338
- TipcConnGetNode function 82
- TipcConnGetNumQueued function 74
- TipcConnGetPid function 82
- TipcConnGetSocket function 72, 141, 143
- TipcConnGetUniqueSubject function 83
- TipcConnGetUser function 83
- TipcConnGetXtSource function 141
- TipcConnGmdFileCreate function 334, 336, 339
- TipcConnGmdFileDelete function 333, 339
- TipcConnGmdMsgDelete function 317, 342
- TipcConnGmdMsgResend function 342
- TipcConnGmdResend function 317, 333, 339, 352
- TipcConnKeepAlive function 151, 152
- TipcConnMainLoop function 117, 141, 142, 144, 334, 338, 339
- TipcConnMsgInsert function 57, 73, 74, 116

- TipcConnMsgNext function 56, 74, 115, 116, 117, 119, 122, 142, 144, 147
- TipcConnMsgProcess function 107, 108, 113, 115, 116, 117, 341
- TipcConnMsgSearch function 56, 74, 116, 117, 118, 119
- TipcConnMsgSearchType function 56, 118
- TipcConnMsgSend function 73, 76, 120, 122, 143, 147, 317, 318, 334, 336, 339
- TipcConnMsgSendRpc function 56, 147, 246
- TipcConnMsgWrite function 121, 122
- TipcConnProcessCbCreate function 107
- TipcConnProcessCbLookup function 107
- TipcConnQueueCbCreate function 109
- TipcConnQueueCbLookup function 110
- TipcConnRead function 73, 74, 76, 115, 118, 119, 123, 124, 317, 334, 338, 342
- TipcConnReadCbCreate function 108
- TipcConnReadCbLookup function 109
- TipcConnSetBlockMode function 74, 106
- TipcConnSetGmdMaxSize function 80, 318, 335
- TipcConnSetSocket function 72
- TipcConnWriteCbCreate function 109
- TipcConnWriteCbLookup function 109
- TipcDispatcher API 248
- TipcDispatcherCreate function 250
- TipcDispatcherCreateDetached function 250
- TipcDispatcherDestroy function 250
- TipcDispatcherDispatch function 250
- TipcDispatcherMainLoop function 250
- TipcDispatcherSrvAdd function 250
- TipcDispatcherSrvRemove function 250
- TipcEvent API 248
- TipcEventCreateConn function 257
- TipcEventCreateMsg function 259
- TipcEventCreateMsgType function 259
- TipcEventCreateSocket function 261
- TipcEventCreateTimer function 262
- TipcEventDestroy function 257, 259
- TipcEventGetCheckMode function 257, 261
- TipcEventGetConn function 257
- TipcEventGetData function 263
- TipcEventGetDispatcher function 257, 259
- TipcEventGetInterval function 262
- TipcEventGetSocket function 261
- TipcEventGetType function 257, 259
- TipcEventSetInterval function 262
- TipcGetGmdDir function 333
- TipcInitCommands function 585
- TipcInitThreads function 139, 140
- TipcMon* function 169
- TipcMon* monitoring functions 143
- TipcMonClientBufferPoll function 382
- TipcMonClientBufferSetWatch function 408
- TipcMonClientCbPoll function 382
- TipcMonClientCpuPoll function 382
- TipcMonClientExtPoll function 382
- TipcMonClientGeneralPoll function 382
- TipcMonClientInfoPoll function 382
- TipcMonClientMsgRecvSetWatch function 408
- TipcMonClientMsgSendSetWatch function 408
- TipcMonClientMsgTrafficPoll function 383
- TipcMonClientMsgTypeExPoll function 383
- TipcMonClientNamesNumPoll function 383
- TipcMonClientNamesPoll function 383
- TipcMonClientNamesSetWatch function 409
- TipcMonClientOptionPoll function 383
- TipcMonClientSubjectExPoll function 383
- TipcMonClientSubscribeNumPoll function 383
- TipcMonClientSubscribePoll function 383
- TipcMonClientSubscribeSetWatch function 409
- TipcMonClientTimePoll function 383, 385
 - using 385
- TipcMonClientTimeSetWatch function 409
- TipcMonClientVersionPoll function 383
- TipcMonExt* function 390
- TipcMonProjectNamesPoll function 382, 611
- TipcMonServerBufferPoll function 383
- TipcMonServerConnPoll function 383
- TipcMonServerConnSetWatch function 409
- TipcMonServerCpuPoll function 384
- TipcMonServerGeneralPoll function 384
- TipcMonServerMsgTrafficExPoll function 384
- TipcMonServerNamesPoll function 384
- TipcMonServerNamesSetWatch function 409
- TipcMonServerOptionPoll function 384
- TipcMonServerRoutePoll function 384
- TipcMonServerStartTimePoll function 384
- TipcMonServerTimePoll function 384
- TipcMonServerVersionPoll function 384
- TipcMonSubjectNamesPoll function 384

- TipcMonSubjectNamesSetWatch function 409
- TipcMonSubjectSubscribePoll function 384, 385
 - using 385
- TipcMonSubjectSubscribeSetWatch function 409
- TipcMsg* function 120
- TipcMsgAck function 317, 338
- TipcMsgAddNamedXmlPtr function 62
- TipcMsgAppendBinaryPtr function 61, 62
- TipcMsgAppendInt2ArrayPtr function 61, 62
- TipcMsgAppendInt4ArrayPtr function 61, 62
- TipcMsgAppendInt8ArrayPtr function 61, 62
- TipcMsgAppendMsg function 60
- TipcMsgAppendMsgArrayPtr function 61, 62
- TipcMsgAppendMsgPtr function 61, 62
- TipcMsgAppendReal16ArrayPtr function 61, 62
- TipcMsgAppendReal4ArrayPtr function 61, 62
- TipcMsgAppendReal8ArrayPtr function 61, 62
- TipcMsgAppendStrArrayPtr function 61, 62
- TipcMsgAppendStrPtr function 61, 62
- TipcMsgAppendUnknown function 63
- TipcMsgAppendXmlPtr function 61
- TipcMsgClone function 20, 56, 59
- TipcMsgCreate function 49, 56
- TipcMsgDestroy function 20, 56, 57, 115, 117, 338
- TipcMsgFieldSetSize function 62
- TipcMsgFileCreate function 65
- TipcMsgFileDestroy function 66
- TipcMsgFileRead function 56, 65
- TipcMsgFileWrite function 64, 65
 - use in debugging 478
- TipcMsgGetCompression function 5
- TipcMsgGetCurrentFieldKnown function 63
- TipcMsgGetDeliveryMode function 10
- TipcMsgGetDeliveryTimeout function 11
- TipcMsgGetDest function 12, 21
- TipcMsgGetExpiration function 13
- TipcMsgGetHeaderStrEncode function 14
- TipcMsgGetLbMode function 15
- TipcMsgGetNumFields function 17, 53
- TipcMsgGetPriority function 18
- TipcMsgGetReadOnly function 19
- TipcMsgGetRefCount function 20
- TipcMsgGetSender function 22
- TipcMsgGetSenderTimestamp function 23
- TipcMsgGetSeqNum function 4, 6, 16, 24
- TipcMsgGetType function 25
- TipcMsgGetUserProp function 26
- TipcMsgIncrRefCount function 20, 56
- TipcMsgNextMsg function 60
- TipcMsgNextType function 58
- TipcMsgNextUnknown function 63
- TipcMsgPrint function
 - use in debugging 478
- TipcMsgPrintError function
 - use in debugging 479
- TipcMsgRead function 54, 60
- TipcMsgSetCompression function 5
- TipcMsgSetCurrent function 52
- TipcMsgSetDelivery function 10
- TipcMsgSetDeliveryMode function 336
- TipcMsgSetDeliveryTimeout function 11
- TipcMsgSetDest function 12, 21, 208
- TipcMsgSetExpiration function 13
- TipcMsgSetHeaderStrEncode function 14
- TipcMsgSetLbMode function 15
- TipcMsgSetNumFields function 17, 57
- TipcMsgSetPriority function 18
- TipcMsgSetSender function 22
- TipcMsgSetSenderTimestamp function 23
- TipcMsgSetType function 25
- TipcMsgSetUserProp function 26
- TipcMsgWrite function 51, 59
- TipcMtCreate function 43
- TipcMtGetCompression function 29
- TipcMtGetDeliveryMode function 30
- TipcMtGetDeliveryTimeout function 31
- TipcMtGetGrammar function 32
- TipcMtGetHeaderStrEncode function 34
- TipcMtGetLbMode function 35
- TipcMtGetName function 36
- TipcMtGetNum function 37
- TipcMtGetPriority function 38
- TipcMtGetUserProp function 39
- TipcMtPrint function
 - use in debugging 478
- TipcMtSetCompression function 28
- TipcMtSetDeliveryMode function 30, 336
- TipcMtSetDeliveryTimeout function 31
- TipcMtSetHeaderStrEncode function 34

- TipcMtSetPriority function 38
 - changing monitoring message priorities 363
- TipcMtSetPriorityUnknown function 38
- TipcMtSetUserProp function 39
- TipcSrv* function 167, 169, 186, 200, 205, 207, 208, 235, 237
- TipcSrv*CbCreate function 205
- TipcSrv*CbLookup function 205
- TipcSrvConn API 248
- TipcSrvCreate function 189, 190, 198, 200, 201, 207, 235, 236, 346, 351, 595
 - shortening connection names 192
- TipcSrvCreateCbCreate function 200
- TipcSrvCreateCbLookup function 200
- TipcSrvDestroy function 200, 202, 236, 238, 346, 351, 595, 600
- TipcSrvDestroyCbCreate function 200, 238
- TipcSrvDestroyCbLookup function 200
- TipcSrvGetConnStatus function 200, 236
- TipcSrvGmdFileCreate function 352
- TipcSrvGmdFileDelete function 200, 352
- TipcSrvGmdMsgServerDelete function 347, 357
- TipcSrvGmdMsgStatus function 348, 349, 358
- TipcSrvGmdResend function 200
- TipcSrvIsRunning function 200
- TipcSrvLogAddMt function 200, 210, 214
- TipcSrvLogRemoveMt function 200, 210, 214
- TipcSrvMsgInsert function 57
- TipcSrvMsgNext function 56
- TipcSrvMsgSearch function 56
- TipcSrvMsgSearchType function 56, 246, 349
- TipcSrvMsgSend function 208, 209, 246
- TipcSrvMsgSendRpc function 56
- TipcSrvMsgWrite function 208, 209
- TipcSrvMsgWriteVa function 208
- TipcSrvPrint function
 - use in debugging 481
- TipcSrvSetAutoFlushSize function 209
- TipcSrvStdSubjectSetSubscribe function 204, 577
- TipcSrvStop function 200
- TipcSrvSubject* function 143
- TipcSrvSubjectCbCreate function 205
- TipcSrvSubjectCbDestroyAll function 243
- TipcSrvSubjectDefaultCbCreate function 205, 242
- TipcSrvSubjectGetSubscribe function 203
- TipcSrvSubjectGmdInit function 354
- TipcSrvSubjectLbInit function 216
- TipcSrvSubjectSetSubscribe function 203, 217
- TipcSrvSubjectSetSubscribeLb function 217, 225
- token ring protocol 85
- Trace_File option 578
- Trace_File_Size option 579
- Trace_Flags option 579
- Trace_Level option 580
- transport protocol 648
- troubleshooting 477
 - Cisco routers 490
 - multicast 487
- tuning rate control 647
- TutCommandParseFile function 189, 498
- TutCommandParseStr function 207, 350, 585
- TutCommandParseStr option 189
- TutCommandParseTypedStr function 585
- TutGetSocketDir function 199
- TutOptionSetUnknown function 214
- TutSocketCheck function 143
- type message property 25

U

- udp_broadcast protocol 193
 - multi-homed RTservers 197
- Udp_Broadcast_Timeout option 580
 - specifying in connection name 193
 - TipcSrvCreate 190
- unalias command 628
- unique subject 22, 162
 - connecting to multiple RTservers 353
 - RTclient using 204
- unique subject connection property 83
- Unique_Subject option 162, 581
 - as connection property 83
 - file-based GMD 331
- unknown field values 63
- unsetopt command 629
- unsubscribe command 630
- unwatch command 631
- user connection property 83

user event

API summary 263

description 263

user-defined message property 26**user-defined message type** 43**user-defined message type property** 39**user-defined subjects** 163**username**

setting for basic security 282

users

basic security 277

setting permissions 279

users command 633**using message files** 65**using wildcards** 160**utilities**

TutCommandParseTypedStr 585

utility functions 174

RTacl 282

TutSocketCheck 143

V**variables**

stop watching 631

watching 634

verbose field type 33**Verbose option** 581**View command, monitoring** 450**W****waiting**

messages 338

waiting for operations to complete 74**warm connection** 201, 202, 235

Server_Auto_Connect option 559

warm connections 307, 346**warm RTclient** 346**warm setting**

Server_Disconnect_Mode option 563

Watch command, monitoring 450**watch command, monitoring** 634**watching**

processing results 411

watching a variable 634

stop watching 631

wildcards

in permissions 281

in subject names 159

in subjects 160

subject examples 160

with publish-subscribe 159

window

displaying output 602

Windows service

starting RTgms 670

starting RTserver 195

write buffer 120

connection property 73, 76

size 73

write callback 120**write callback connection property** 109**write connection callback** 336**write mutex connection property** 81**write operations** 152**write timeout**

getting existing settings 78

setting the value 78

write timeout connection property 77**write timeouts**

disabling 77

X

X.500 297

Xt Intrinsic 141

Xt Intrinsic functions

 TpcConnGetXtSource 141

 XtAppAddInput 141, 143

 XtAppAddTimeOut 142

 XtAppAddWorkProc 142

 XtAppMainLoop 141, 143

 XtAppPending 142

 XtAppProcessEvent 142

 XtRemoveInput 141

XtAppAddInput Xt Intrinsic function 141, 143

XtAppAddTimeOut Xt Intrinsic function 142

XtAppAddWorkProc Xt Intrinsic function 142

XtAppMainLoop Xt Intrinsic function 141, 143

XtAppPending Xt Intrinsic function 142

XtAppProcessEvent Xt Intrinsic function 142

XtRemoveInput Xt Intrinsic function 141

Z

Zero_Recv_Gmd_Failure option 582

ZLIB

 for message compression 271