TIBCO SmartSockets™

Java Library User's Guide and Tutorial

Software Release 6.8
July 2006

The Power of Now®

Important Information

Contents

Figures

Tables

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.

Topics

About This Book

This reference provides the detailed information you need to use and develop distributed applications with the SmartSockets Java class library. This guide also contains a tutorial to help you quickly learn to use the SmartSockets Java class library. Before starting the tutorial, install SmartSockets and the SmartSockets Java class library. Installation instructions for SmartSockets can be found in the *TIBCO SmartSockets Installation Guide*.

This guide is intended to be a supplement to the *TIBCO SmartSockets User's Guide*. Many key concepts are explained in detail there and are the same for both the Java and C application program interfaces (APIs). This guide gives a brief overview of SmartSockets, emphasizing the differences between the Java and C APIs.

For detailed reference information on the SmartSockets Java classes, see the online reference information, provided in JavaDoc format with the SmartSockets product. The *TIBCO SmartSockets Installation Guide* tells you where to find those files. For an overview of the new features, changes, and enhancements in this Version 6.8 release, see the *TIBCO SmartSockets Release Notes*.

Intended Audience

This guide is for software developers and project managers who want to know how SmartSockets and the SmartSockets Java class library can help them build distributed applications with program-to-program communication.

Some prerequisite knowledge is needed to understand the concepts and examples in this guide:

- working knowledge of Java
- familiarity with the operating system is required for developing SmartSockets applications (UNIX, Windows, OpenVMS, or whatever platform is running SmartSockets)

  This includes knowing how to log in, log out, edit a text file, change directories, list files, and build and run a program.
- understand general messaging and publish/subscribe concepts and terminology
- understand the SmartSockets messaging and publish/subscribe concepts described in the *TIBCO SmartSockets User's Guide*

Related Documentation

For more information about TIBCO SmartSockets, see:

- *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) reference 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
`monospace`	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.
`monospace bold`	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.
Italic	Italic text is used as follows: • In code examples, file names etc., for text that should be replaced with an actual value. For example: "Select *install-dir*/`runexample.bat`." • For document titles. • For emphasis.
Bold	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. Submenus and options of a menu item are indicated with an angle bracket, for example, **Menu > Submenu.**
	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 *Italic*) 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 Introducing TIBCO SmartSockets

TIBCO SmartSockets is an interprocess messaging software product that enables processes to communicate quickly, reliably, and securely across different operating system platforms. The communicating processes can reside on the same machine, on LANS, on WANs, or anywhere on the Internet. SmartSockets is an industrial-strength package that takes care of network interfaces, guarantees delivery of messages, handles communication protocols, and deals with recovery after system or network failures. The SmartSockets' programming model is built specifically to offer high-speed interprocess communication, scalability, reliability, and fault tolerance. It supports a variety of communication paradigms, including publish-subscribe, peer-to-peer, and request-reply. Included as part of the package are graphical tools for monitoring and debugging your distributed applications.

The term message is used throughout this manual. It should not be confused with the universally known concept of an email message. A SmartSockets message is a structured packet of information that is transferred between two or more programs, which may or may not reside on the same machine. It is not unusual for a SmartSockets message to exist only in memory and never be written to disk. A message is a mechanism that enables program-to-program communication to occur in a manner easily understood by both you and the programs.

Topics

What Comprises TIBCO SmartSockets?

TIBCO SmartSockets consists of a suite of programming interfaces and class libraries, ready-to-run programs, source code for sample programs, and extensive documentation. It is designed to get your network programs running as quickly as possible.

These components are part of the standard SmartSockets distribution:

- SmartSockets Application Programming Interface (API) provides a C-callable library of functions for communicating between programs and monitoring your distributed applications.
- SmartSockets Java Class Library provides classes, objects, and interfaces to Java applications, allowing them to leverage the functionality of the SmartSockets API.
- SmartSockets C++ Class Library provides an object-oriented layer on top of the standard SmartSockets services.
- RTserver, a powerful software message router, empowers applications with the publish-subscribe communications model.
- RTmon, a powerful tool for monitoring and debugging your distributed project, is accessible through a GUI and also through a command-line interface.
- Structured message types, a message with predefined field types, enable transparent data conversion. SmartSockets comes out of the shipping box with many predefined message types to get you working quickly. You can easily extend these by defining your own types.
- Options and Command Language enable you to reconfigure your SmartSockets applications easily.
- Sample programs get you off to a fast start.
- Documentation is available in printed and electronic format, where it can be viewed from any Internet browser.

TIBCO SmartSockets Features

Programs built with SmartSockets require fewer lines of code than those constructed with other IPC mechanisms and have sophisticated added benefits. SmartSockets provides guaranteed message delivery (GMD), ensuring that your application's data is delivered to the component processes in a completely reliable manner. SmartSockets also provides fault-tolerance capabilities that make it easy to develop robust systems. The key features are:

- Insulates you from complexities of network programming:
  - interoperability in heterogeneous computing environment
  - transparent multiple protocol support
  - automatic data conversion across heterogeneous platforms
  - location transparency; sending process(es) need not know location of receiving process(es)
  - thread-safe API
  - multi-threaded servers
- Publish-Subscribe communications services:
  - hierarchical namespace
  - synchronous / asynchronous message transfer
  - both peer-to-peer and client-server models
  - high-speed message routing
  - one-to-many, many-to-one message transfer
  - both wildcard publishing and wildcard subscribing
  - load balancing
  - message compression
- Graphical monitoring and administration:
  - animated graphical tree of your application with real-time updates
  - graphical view of RTserver connectivity and IPC traffic
  - point-and-click interface to program and IPC information
  - watch events as they happen
  - poll for information at regular intervals
  - monitoring is nonintrusive; application does not need to be modified

  - monitor multiple processes simultaneously
- Prioritized message queues:
  - messages can be processed first-in, first-out (FIFO) or in priority order
  - message queue can be searched for messages of a given type before reading
  - high-priority messages can be placed in the front of the queue
- Guaranteed message delivery:
  - messages are delivered even in the event of network or system failure
  - guaranteed delivery can be specified on a message-by-message basis
  - guaranteed delivery can be specified on all messages of a given type
- Fault tolerance:
  - dynamic message routing across any system topology
  - automatic detection and recovery from network / system failure
  - hot failover of clients and servers
  - auto start and restart of programs
  - keep-alives (heartbeats)
  - read / write / connect timeouts
  - flexible message buffering for both senders and receivers
  - multiple RTservers
- Structured messages:
  - typed messages; types are defined using concise message grammar
  - messages contain data and properties, such as sender, priority, and delivery mode
  - extensive API to create, construct, duplicate, and access messages
  - reusable and extensible message types
  - messages can exist within another message
  - message fields can be accessed sequentially or by name
  - messages can contain XML data

- Robust and professional set of programming tools:
  - — object-oriented API with extensive set of callbacks
  - — reusable Java and C++ classes
  - — program and IPC traffic monitoring and debugging tools
  - — message logging
  - — settable options that require no programming
  - — commands (including commands that set options) that can be placed in text files or entered interactively
  - — upward compatibility as you upgrade to new versions of SmartSockets
  - — support for Microsoft ActiveX programming environments
  - — native Java support using SmartSockets Java class library
- Simple installation and configuration:
  - — does not require root or SYSTEM privilege to install
  - — does not require a special process or daemon on every machine
  - — does not require any modifications to the operating system kernel
- Security services:
  - — Basic Security using usernames and passwords, and permissions lists in RTserver
  - — message filtering through a gateway process

For information on the new features and options available with this release of SmartSockets, see the *TIBCO SmartSockets Installation Guide*.

Java Message Service

In addition to using SmartSockets with Java, you can use the Java Message Service (JMS). For more information about TIBCO SmartSockets JMS, contact your TIBCO sales representative or TIBCO Product Support.

Platform Support

SmartSockets is supported on a number of different computing platforms, including many types of UNIX and Windows, as well as most platforms that support Java. This list is expanded frequently and others may be available. Contact TIBCO Software Inc. for more information.

Source Code Availability

The SmartSockets Java class library is shipped as a Java Archive (JAR) containing the classes and interfaces necessary for building Java applications that utilize SmartSockets. Most other parts of SmartSockets are shipped as executables or as object-code libraries.

Programming Language Support

SmartSockets is designed to integrate effortlessly with Java, C, C++, and ActiveX environments. However, any language that supports a C or C++ binding can effectively use SmartSockets.

Major Components of TIBCO SmartSockets

The major components of SmartSockets are:

Messages	are the packets of information sent between processes.
Message types	are the templates that describe the data part of a message.
Connection	is an endpoint of a communication link used to send and receive messages between two processes.
RTserver	is a process that extends the features of connections to provide transparent publish-subscribe message routing among many processes.
RTclient	is any program that connects to RTserver and uses its services (under this definition RTmon can be considered an RTclient).
RTmon	is a powerful tool for monitoring and debugging your distributed project. RTmon allows you to use a graphical point-and-click interface for watching things like IPC traffic and process information. RTmon is also accessible through a command-line interface.

Messages

Within a SmartSockets application, interprocess communication occurs using 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:

- graphical commands, such as changing the color of an object or popping up a view on a specified display
- commands to a process' command interface
- images or audio
- user-defined binary data, such as C data structures or Java objects
- process information, such as the names and the number of processes currently subscribed to a particular subject
- IPC traffic information, such as how many messages are currently in the message queue of a program
- executable programs

All of these 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.

Message Composition

A message is composed of the header and the data. The header contains properties that specify control information about the message. Examples of SmartSockets message properties are the message sender, destination, type, priority, and delivery mode. The data contains the information you wish to send and is usually the largest part of the message. The message type property defines the structure of the data part of the message.

Working with Messages

Typically, when building a SmartSockets application, these steps are required when constructing a message:

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

The same message can be used many times, changing only the data part of the message or a property such as the destination. There are many different types of fields that can be appended to a message's data. These field types include three sizes of integers, character strings, three sizes of real numbers, and arrays of the scalar field types, such as an array of four-byte integers. Fields can also be associated with a name, allowing them to be accessed by that name. Messages themselves can even be used as fields within other container messages; this allows operations such as large transactions to be represented with a single message.

Once a message is constructed, it can be sent to another process through a connection or published to a subject to be delivered to multiple processes.

Message Types

As described earlier, each message has a type property that defines the structure of the data property of the message. A message type can be thought of as a template (or 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 is then available for use as the type for any number of messages.

SmartSockets provides a large number of standard predefined 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 your own user-defined message types. 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.

Connections

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.

RTserver and RTclient

While connections allow two processes to send messages to each other, RTserver and RTclient allow many processes to communicate with each other. RTserver routes messages between RTclients. A key feature of SmartSockets is the ability to distribute RTservers and RTclients anywhere over a network. Different processes can be run on different computers, taking advantage of all the computing power a network has to offer. Processes can be dynamically started and stopped while the system is running.

The functionality of RTserver and RTclient is layered on top of connections and messages, but adds greater functionality and ease of use. Some of these features are listed below.

- RTserver and RTclient simplify setup and control through options that require no programming.
- RTserver can partition a group of RTclients into a project.
- RTserver and RTclient use logical addresses called subjects for the sender property and destination property of messages.
- RTserver and RTclient use a publish-subscribe communications model, allowing a program to send a message to multiple receivers with a single operation.
- Messages can be dynamically routed through a network of RTservers using a lowest cost algorithm.
- Messages can be compressed to conserve bandwidth; message compression is useful in situations where you need to transfer messages across lower bandwidth connections, such as WANs and wireless networks.
- Multiple RTservers can distribute the load of message routing.
- Messages can be uniformly distributed to a series of RTclients through load balancing.
- An RTclient can continue running when RTserver is temporarily unavailable, and even attempt to reconnect to other RTservers which may still be operating.
- Through RTserver, program and IPC traffic information can be monitored by an RTclient and also through the RTmon GDI.
- An RTclient can monitor data created by one or more other RTclients, referred to as extension data, by directly polling the other RTclients; the RTserver is not involved in creating this kind of monitoring data.
- RTserver and RTclient can use callbacks to execute user-defined functions when certain operations occur.

- RTserver automatically converts messages sent between different types of computers.
- Messages can have guaranteed message delivery, which enables total recovery from network failures.
- RTclient and RTserver are inherently thread-safe.

RTserver and RTclient Composition

Before you use RTserver and the RTclient API, you should have an understanding of the concepts involved. The RTserver and RTclient architectures and the major concepts that you need to understand are:

RTserver	is a process that extends the features of connections to provide transparent publish-subscribe message routing among many processes.
RTclient	is any program that connects to an RTserver and accesses its services (under this definition RTmon is considered an RTclient).
Project	is a group of RTservers and RTclients working together.
Subject	is a logical address for a message; RTclient subscribes to, or registers interest in, subjects; an RTclient also publishes or sends messages to subjects.
Monitoring	allows you to examine detailed information about your project in real time.

Projects

A SmartSockets project is a group of RTclients working together with one or more RTservers to perform some set of tasks as part of a specific system. Within a project, processes can communicate with other processes on the same machine or over the network. RTclient processes in different projects cannot send messages to each other.

Typically, an RTclient belongs to only one project. An RTserver does not belong to any project, but can provide message routing services for one or more projects. You can think of a project as a firewall that prevents messages from being dispatched outside the specified process group. It is possible for an RTclient to connect to more than one project in the same RTserver or to multiple projects across RTservers. See the *TIBCO SmartSockets User's Guide* for more information.

For example, in Figure 1 the RTclients are running on the same network and are each monitoring two factories, so the projects named FAC1 and FAC2 are used to ensure that messages are not sent between the two separate projects. The option Project is used to specify the project to which an RTclient belongs. The default value for Project is `rtworks`. Always set this option to prevent becoming part of the default project, which can cause unwanted messages to be received.

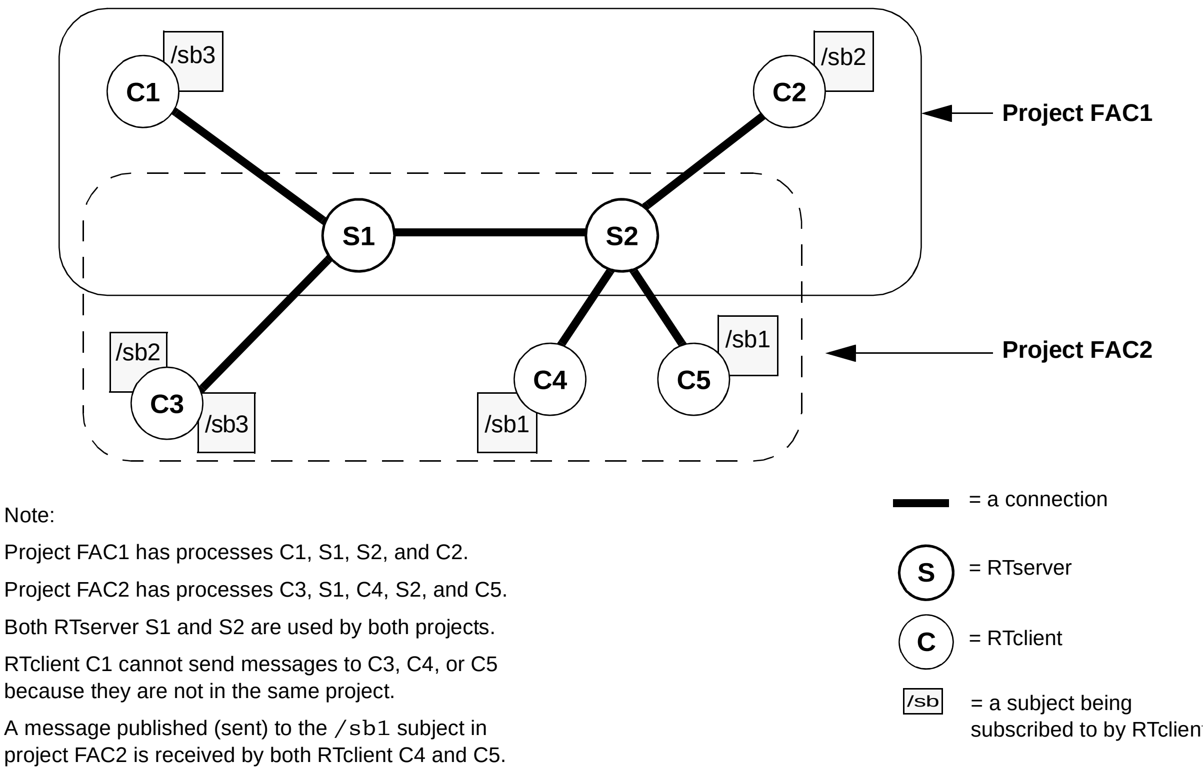
Subjects

Just as projects restrict the boundaries of where messages are sent, subjects also further partition the flow of messages within a project. A subject is the fundamental concept used in taking advantage of SmartSockets publish-subscribe communication services. A subject is a logical message address that provides a virtual connection between RTclients.

When an RTclient has subscribed to a subject, it gets any messages sent to RTserver whose destination property is set to that subject. For example, in a stock trading application, you might partition messages by stock market sectors, such as computer stocks, automobile stocks, financial stocks. These areas would be declared as subjects such as `/stocks/computer`, `/stocks/auto`, `/stocks/financial`. All messages pertaining to computer stocks are constructed with the `/stocks/computer` subject as their destination property. Any RTclient interested in receiving messages published to `/stocks/computers` subscribes to the `/stocks/computer` subject. This is also known as the publish-subscribe communications model in that the RTclients publish messages to a specific subject, and the RTclients subscribe to subjects in which they are interested.

If you are not using SmartSockets publish-subscribe services, when two processes create connections to each other, they need protocol-specific network addresses to begin communicating (for example, TCP/IP needs a node name and port number). If a process wants to send a message to many other processes, it first needs to know the protocol-specific network addresses of the other processes and then creates connections to all of those processes. This kind of architecture does not scale well as configuration is complicated and tedious. The RTserver/RTclient architecture's use of subjects for message addresses allows 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 receiving that subject.

Figure 1 RTserver and RTclient Architecture



RTserver

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 powerful message router that uses connections to make large-scale distributed IPC easier.

In addition to routing 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 sent are routed between processes on the same node, then the placement of an extra RTserver on the local node can reduce the consumption of network bandwidth (processes on the same node can use the non-network local IPC protocol).

Currently, native Java clients cannot make use of the local IPC protocol.

RTclient

An RTclient is a process that is connected to RTserver as a client. Usually, each RTclient has exactly one connection to exactly one RTserver. This is a single, global RTserver connection. An RTclient can send messages, receive messages, and create callbacks using this connection, just as it does any other connection. The message routing capabilities of RTserver are transparent to 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. 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 can have multiple RTserver connections. Usually, the single, global RTserver connection is sufficient, but when threads such as Java applets or servlets need individual connections, you can create new RTserver connections independent of the global RTserver connection. For more information, see Multiple RTserver Connections on page 27.

RTmon

RTmon is a powerful tool you can use to monitor and manage your distributed project. You can access the RTmon through 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 interface that is intuitive and easy to use. The RTmon provides an assortment of tools for viewing your project, including graphical trees, browsers, graphical charts, and watch windows. The RTmon CI is a command-line-based interface, allowing you to monitor and manage your project using a command prompt.

The RTmon GDI has been deprecated and may be removed in a future release.

The RTmon Main window presents an animated graphical tree of your project, with nodes in the tree representing RTclients, the subjects to which they are subscribing, and the RTservers running in the project. As changes occur, the tree is updated in real time. The Watch Server Connections window graphically displays the RTservers in your project and their connection topology. You can use this window to monitor the load on your RTservers and their connections using a variety of different metrics. RTmon even allows you to look at individual messages and their contents as they are passed from process to process.

One of the principal features of RTmon is that it is non-intrusive. You can monitor, debug, and log information in your project without changing the running processes. RTmon also provides real-time system usage information on processes, such as CPU and memory resources. This is useful for stopping a process that is using excessive system resources.

The content and functionality of the RTmon GDI are identical between the Windows and Motif platforms.

Chapter 2 Lesson Overview

This chapter introduces you to your SmartSockets Java class library lessons, which help you quickly start using the Java class library. These lessons show you how simple it is to use SmartSockets to build, test, and debug a distributed application consisting of a number of programs communicating with one another using messages. Once you complete the exercises in these lessons, you will understand how SmartSockets makes your job of network programming much easier.

Topics

Before You Begin

Before you can start the lessons, you must have SmartSockets and the SmartSockets Java class library installed on your system. The complete information for installing SmartSockets and the SmartSockets Java class library is in the *TIBCO SmartSockets Installation Guide* shipped with your order. Be sure to check the online README file for any last minute changes or corrections.

Required Software

The SmartSockets Java class library is compatible with SmartSockets 5.0 or higher. In addition, the Java class library was built using Java2. You must have Java 2 Software Developer Kit (JSDK), Version 1.3 or higher, to develop SmartSockets Java programs.

Including the Java Class Libraries

Remember that to include SmartSockets Java class libraries, you must specify the path to those class files in your development environment.

On Solaris using the C shell, set your environment variable using:

```
setenv CLASSPATH $RTHOME/java/lib/ss.jar:{$CLASSPATH}
```

For other shells, see your operating system documentation.

On Windows NT, edit the `%RTHOME%\bin\i86_w32\ssvars32.bat` batch file by adding a semicolon to the end of the current path and append the path to the `ss.jar` file:

```
%RTHOME%\java\lib\ss.jar
```

TIBCO SmartSockets Java Class Library Scope

While much of the functionality provided by the SmartSockets C language API is made available to Java programs with the SmartSockets Java class library, not every feature has been ported to Java. For up-to-date details of which specific functionality is not yet present in the SmartSockets Java class library, see the online JavaDoc format reference information. For the exact location of these files in the distribution, see the *TIBCO SmartSockets Installation Guide*.

Using the Java Class Library

The fastest way to learn SmartSockets is by example. Before many of the key concepts of SmartSockets are introduced, the first lesson demonstrates how to write, compile, and execute two sample Java programs that use SmartSockets for interprocess communication.

The *TIBCO SmartSockets User's Guide* describes SmartSockets in much greater detail, using a layered approach: first describing messages, then peer-to-peer connections, then client-server connections, and finally how to monitor and debug a SmartSockets application. When concepts are not clear from the tutorial, or are not presented in enough depth, refer to the *TIBCO SmartSockets User's Guide* for more details.

The Java Class Library Lessons

These are the lessons on using the SmartSockets Java class library. To best learn how to use the Java class libraries, remember to do the lessons in order, and do not skip any of the lessons. Begin the lessons, in this order:

- Lesson 1: Your First Program
- Lesson 2: Publish-Subscribe
- Lesson 3: Messages
- Lesson 4: Callbacks
- Lesson 5: TIBCO SmartSockets Options
- Lesson 6: Java Applets

After you have completed the lessons, you will better understand the more advanced information in the remaining chapters:

- Chapter 9, RTclient Options
- Chapter 10, Using Java Clients
- Chapter 11, Guaranteed Message Delivery

Chapter 3 Lesson 1: Your First Program

In this lesson you learn about:

- how to use the SmartSockets Java classes
- how to write a program to send a message
- how to write a program to read a message
- how to compile and run SmartSockets Java programs

Topics

Lesson 1 Overview

The files for this lesson are located in the directories:

Windows:

```
%RTHOME%\java\tutorial\lesson1
```

UNIX:

```
$RTHOME/java/tutorial/lesson1
```

During this lesson, you write, compile, and run these programs:

- `send` — sends a message with the text "Hello World!" contained in its data part
- `receive` — reads a message and prints out the data part of the message (in this example, this is "Hello World!")

To use the SmartSockets Java classes effectively, it is important that you understand these functional areas that each class manages:

- SmartSockets classes start with a T prefix
- IPC classes start with Tipc (the ipc is for interprocess communication)
- methods that manipulate messages are in the TipcMsg class
- methods that manipulate message types are in the TipcMt class
- methods that manipulate connections are in the TipcConn class
- methods that communicate with RTserver are in the TipcSrv class
- utility methods are in the Tut class
- methods allowing a client to monitor other clients and servers are in the TipcMon class

A Hello World! Program

In this section, the complete source code for your first SmartSockets Java program is presented. Be sure SmartSockets and the SmartSockets Java Class Library are installed properly on your system.

The files for this lesson are located in the directories:

Windows:

`%RTHOME%\java\tutorial\lesson1`

UNIX:

`$RTHOME/java/tutorial/lesson1`

Step 1 **Create a working directory**

Before you begin writing your first program, create a working directory where you have read and write access to store the examples.

Step 2 **Copy the tutorial files**

Copy the tutorial files from the `lesson1` directory into your working directory.

Line numbers appear on the far left margins of code examples. Note that these numbers are not part of the program but are used to refer to different lines in the source code. This is the `send.java` program:

//---
// Program 1: send.java

```
1 import java.io.*;
2 import com.smartsockets.*;

3 public class send {

4   public static void main(String[] argv) {
5     TipcSrv srv = TipcSvc.getSrv();
6     TipcMsg msg = TipcSvc.createMsg(TipcMt.INFO);
7     msg.setDest("/ss/tutorial/lesson1");
8     msg.appendStr("Hello World!");

      try {
9       srv.send(msg);
10      srv.flush();
11      srv.destroy();
12    } catch (TipcException te) {
13      Tut.warning(te);
      } // catch
    } // main
  } // send
```

It might be difficult to believe that a complete SmartSockets program can be contained in so few lines; this is one of the main benefits of SmartSockets. Hundreds of lines of interprocess communication code (such as sockets or RPCs) can be reduced to just a few lines of SmartSockets Java code.

Let's take a look at the key lines of this program:

Line 2 The SmartSockets Java package, `com.smartsockets`, is imported. This step is required.

Line 5 A reference to the RTserver object is placed in the `srv` object. Using TipcSvc.getSrv() allows a single global RTserver connection to be created when needed.

Line 6 A TipcMsg object (`msg`) is created using the INFO message type.

Line 7 Sets the subject to which the message is being published. In this case the subject is `/ss/tutorial/lesson1`.

Line 8 The text message `"Hello, World!"` is appended as the first data field in the `msg` message object.

Line 9 The message is sent to RTserver using the `send` method. Note that the `send` method automatically created a connection to RTserver, which must already be running.

Line 10 The connection to RTserver is flushed, ensuring the message is sent immediately.

Line 11 The connection to RTserver is closed. It is a good practice that all programs sending data to RTserver destroy their connection when it is no longer needed.

The SmartSockets methods referred to in this program are:

- TipcSvc.getSrv()
- TipcSvc.createMsg()
- TipcMsg.setDest()
- TipcMsg.appendStr()
- TipcSrv.send()
- TipcSrv.flush()
- TipcSvc.destroy()

From the API naming conventions, you can see that the methods TipcSrv.send and TipcSrv.flush are used to communicate with RTserver, because they are part of the TipcSrv class.

Compiling

Once the program has been written, it must be compiled with an appropriate Java compiler for your platform. The examples in this manual are compiled with Sun Microsystems Java Development Kit.

Step 3 **Compile the sending program**

To compile the `send.java` program, use this command:

```
$ javac send.java
```

Once compiled, the sending program is ready to run. Before running it, however, you need to create a second program, `receive.java`, to read and print the message that you are going to send using the `send` program.

A Program to Read a Message

The next program, `receive.java`, reads and prints out the contents of the message being published from the `send` program described in A Hello World! Program.

Step 4 **Enter or copy the receive.java program**

As before, enter this program interactively using your favorite editor, or copy it from the `receive.java` file.

This is the `receive.java` program:

```
//-----------------------------------------------------
// Program 2: receive.java

import java.io.*;
import com.smartsockets.*;

public class receive {

  public static void main(String[] argv) {
    TipcMsg msg = null;
    String text = null;
```

```
7      TipcSrv srv = TipcSvc.getSrv();
       try {
8        srv.setSubjectSubscribe("/ss/tutorial/lesson1", true);
9        msg = srv.next(TipcDefs.TIMEOUT_FOREVER);

10       msg.setCurrent(0);
11       text = msg.nextStr();
12     } catch (TipcException e) {
13       Tut.fatal(e);
       } // try-catch

14   System.out.println("Text from INFO message = " + text);
     } // main
   } // receive
```

As with the sending program, notice how the receiving program consists of so few lines of code. Compare this with a similar program written using pipes, sockets, or shared memory. SmartSockets programs are typically much shorter than those developed with traditional low-level technologies and are instantly able to leverage the power of the publish-subscribe paradigm.

Let's take a look at the key lines of this program:

Line 2 The SmartSockets Java package is imported. This step is required.

Line 7 A reference to the RTserver object is placed in the `srv` object.

Line 8 Subscribing to the `/ss/tutorial/lesson1` subject allows receipt of messages published by the `send` program.

Line 9 The `srv` object's `next` method is used to wait for a message to be received. This line blocks forever, as indicated by the TipcDefs.TIMEOUT_FOREVER parameter. (The `next` method takes only one parameter, the time in seconds to wait.)

Line 10 Now that a message has been received into the `msg` object, the setCurrent method is used to position the field pointer. Note that the first data field of the message is specified by the value `0`.

Line 11 The string data contained in the message is extracted with the nextStr method and copied into the `text String` object.

Line 14 The text data is printed on the console with the println method.

Step 5 Compile the receiving program

After you write the program, you need to compile it:

```
$ javac receive.java
```

Running the Application

Now that both the sending and receiving programs have been created and compiled, you can run the complete application to see if the message is successfully transmitted.

Step 6 **Start the RTserver**

Two windows need to be open, both set to the working directory, to see the application properly. In one window, start RTserver:

```
$ rtserver -check
```

On platforms that support both 32- and 64-bit, use the `rtserver64` command to run the 64-bit version of the `rtserver` script.

Specifying the `-check` argument starts the non-optimized version of RTserver, which performs additional validation and checking. The optimized version is faster because there is no checking, but it is much harder to diagnose a problem. During all your development and testing, you should run RTserver with the checking turned on. Even in your production environment, you might prefer to run the check version of RTserver. The optimized version is best for enterprise applications where speed is the most important factor.

Step 7 **Start the sending program**

Start the sending program in the other window:

```
$ java send
```

Step 8 **Start the receiving program**

To read and output the message, start the receiving program using this command in the second window:

```
$ java receive
```

The receiving program is waiting for a message. This is because the sending program was executed first. It sent its message, and because the receiving program had not yet been started, there were no subscribers wishing to receive the message. RTserver does not send out messages if there are no processes available to receive them.

Step 9 **Start the sending program again**

This time, start the sending program while the receiving program is already running. Go back to the first window and re-execute the sending program. (Remember, the receiving program is still running and waiting for the message.)

```
$ java send
```

This output is displayed in the window where the receiving program is running:

```
Text from INFO message = Hello World!
```

This indicates the message was read, and its field was accessed and displayed properly.

An important lesson here is that synchronizing processes at startup is critical. Make sure your receiving processes are started first. This is a common error for first-time developers of network programs.

In just a short time, you have written your first successful SmartSockets application in Java!

What's Going On

Notice that nowhere do we call the constructor for TipcSrv or TipcMsg, despite the obvious fact that instances of each class are being created. Instead, the TipcSvc class is used to "get" instances of each. Looking at the online reference, notice that TipcSrv and TipcMsg aren't classes at all. They are interfaces. Why aren't we creating an instance of TipcSvc?

This concept is known as the abstract factory pattern. In this model, instances of classes aren't created directly. Instead, a factory class is used to create them indirectly. TipcSvc is that factory class. You never have to create an instance of TipcSvc because all of the methods in it are static. When you create a message with TipcSvc.createMsg, it creates an instance of a non-public class that implements the TipcMsg interface and returns a reference to that class to your program, which you manipulate through the abstract TipcMsg interface. The abstract factory model allows SmartSockets developers more freedom in altering the structure of the library without impacting end-user code. The TipcMsg interface could be implemented by several classes, or only one. The inheritance hierarchy can be rearranged, and classes could be removed or renamed, without affecting your code. Indeed, due to Java dynamic linking, you shouldn't even have to recompile your code when such changes are made.

Also note that the online reference does not list the methods `send` or `flush` under TipcSrv. This is because TipcSrv extends TipcConnClient, and most of the methods dealing with sending and reading messages are handled by that class.

Finally, it's important to note that TipcSvc.getSrv does not actually create a connection to RTserver. SmartSockets uses a "lazy" scheme when making connections, deferring the process until the connection is actually required. For the sender, this doesn't happen until `flush` is called. The receiver creates a new connection when `next` is called. You can also explicitly create a new connection with the TipcSrv.create method.

Multiple RTserver Connections

Usually, as shown in the code for A Hello World! Program, you use TipcSvc.getSrv to create a single global RTserver connection. In some cases, you may need to create multiple RTserver connections from your RTclient:

- threads such as Java applets or servlets might require individual connections to register independent subscriptions and callbacks
- threads such as Java applets or servlets might require individual connections for proper remote procedure call (RPC) handling
- topology bridges between two or more RTserver clouds might need to be created to satisfy the needs of a large scale enterprise system

To create multiple RTserver connections, use the TipcSvc.createSrv method. This creates new RTserver connections independent of the global RTserver connection created by TipcSvc.getSrv. TipcSvc.createSrv allows properties to be associated with a TipcSrv object so that each RTserver connection can have its own option settings.

Here is an example of a program that creates multiple RTserver connections:

```
//-----------------------------------------------------------
// multi.java -- Java application with multiple connections

import com.smartsockets.*;

public class multi {
  public static void main(String[] argv) {
    TipcMsg msg = null;
    String text = null;

    try {
      // the sender connection
      TipcSrv sender = TipcSvc.createSrv();
      sender.setOption("ss.unique_subject", "sender");

      // the receiver connection
      TipcSrv receiver = TipcSvc.createSrv();
      receiver.setOption("ss.unique_subject", "receiver");

      // create the message
      msg = TipcSvc.createMsg(TipcMt.INFO);
      msg.setDest("/multi");
      msg.appendStr("Hello, World!");

      // subscribe the receiver connection to subject "/multi"
      receiver.setSubjectSubscribe("/multi", true);
      receiver.flush();
```

```
        // send the message over the sender connection, then close the connection
        sender.send(msg);
        sender.flush();
        sender.destroy();

        // the receiver connection will now receive the message
       TipcMsg receivedMsg = receiver.next(TipcDefs.TIMEOUT_FOREVER);
       receivedMsg.setCurrent(0);
       text = receivedMsg.nextStr();
      }
      catch (TipcException e) {
        Tut.fatal(e);
      } // try-catch

      System.out.println("Text from INFO message = " + text);
    } // main
  } // multi
```

Error Handling

One nice feature of Java is the enforced error-handling capability provided by the exception mechanism. The SmartSockets Java Class Library fully utilizes exceptions, throwing them to indicate many types of error conditions. As illustrated by the example sending and receiving programs above, an important part of any SmartSockets Java program is appropriate error-handling procedures in the `catch` blocks following code that may throw an exception. The online class library documentation details the exceptions that each method can throw, as well as explanations of common error conditions.

Specific classes of exceptions should be caught when possible, instead of using the blanket Exception class. All SmartSockets exceptions are inherited from the TipcException class, which is itself derived from Exception. As an example of the appropriate way to catch exceptions, see these examples:

```
try {
  srv.send(a_message);
  // possibly more code
} catch (TipcException err) {
  // handle the error here
}
```

The above specifies the exact exception that may be thrown, TipcException. This is preferred over the more generic code:

```
try {
  srv.send(a_message);
  // possibly more code
} catch (Exception err) {
  // handle error
}
```

Using the more specific exception object makes debugging easier and more succinct, because other code in the `try` block is likely to generate different specific exceptions.

Summary

The key concepts covered in this lesson are:

- Reliable interprocess communication can easily be added to your program with the SmartSockets Java classes.
- SmartSockets uses a consistent naming convention for its classes, making it easy to understand and locate the functionality you need.
- All SmartSockets programs must import the `com.smartsockets` package.
- With very few lines of code you were able to create a program to send a message. With a few more lines of code you were able to write a program that reads a message and outputs it (these programs also perform a number of other valuable tasks that are covered in later lessons).
- Before publishing a message, be sure that your receiving program is running.
- Many SmartSockets methods potentially throw an exception. These exceptions should be handled appropriately within `catch` blocks in case the method did not complete correctly.
- SmartSockets programs should always try to catch the TipcException or TipcException-derived exception thrown by a block of code in preference to the more generic Exception class, to help differentiate SmartSockets errors.

Chapter 4 Lesson 2: Publish-Subscribe

In this lesson you learn about:

- how the SmartSockets publish-subscribe model works
- what an RTserver is and how to run it
- what a project is
- what a subject is
- how to send one message to many processes in a single operation

Topics

Lesson 2 Overview

The files for this lesson are located in the directories:

Windows:

```
%RTHOME%\java\tutorial\lesson2
```

UNIX:

```
$RTHOME/java/tutorial/lesson2
```

In the previous lesson, you wrote a program to send a message and a second program to read and print out the message. This message was not transferred directly from sender to receiver; rather, it went first from the sender to the SmartSockets RTserver, and then from RTserver to the RTclient program (in this case, the receiver). Notice that the intervention of RTserver is completely transparent. In fact, RTserver performs many important tasks transparently. Many of these tasks are described in this lesson.

SmartSockets does allow you to send messages directly between two RTclients, peer-to-peer, without using RTserver. This is accomplished using connections. The SmartSockets TipcConn class is used to work with peer-to-peer connections. The lessons in this reference focus on using RTserver for interprocess communication, because this is the method most commonly used and it offers numerous advantages over SmartSockets peer-to-peer connections. For detailed information on connections, see the *TIBCO SmartSockets User's Guide*.

What is RTserver?

While connections allow two processes to send messages to each other, RTserver allows multiple RTclients to communicate easily. RTserver routes messages between RTclients. RTserver can be thought of as a message switch—a large software switch for messages.

A key feature of SmartSockets is the ability to distribute RTservers and RTclients over a network. Different processes can be run on different computers, taking advantage of all the computing power a network's systems have to offer. RTservers and RTclients alike can be dynamically started and stopped while the system is running.

The functionality of RTserver and RTclient is layered on top of connections and messages, but adds capability and ease of use to these functions. 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.

Distributing Message Load

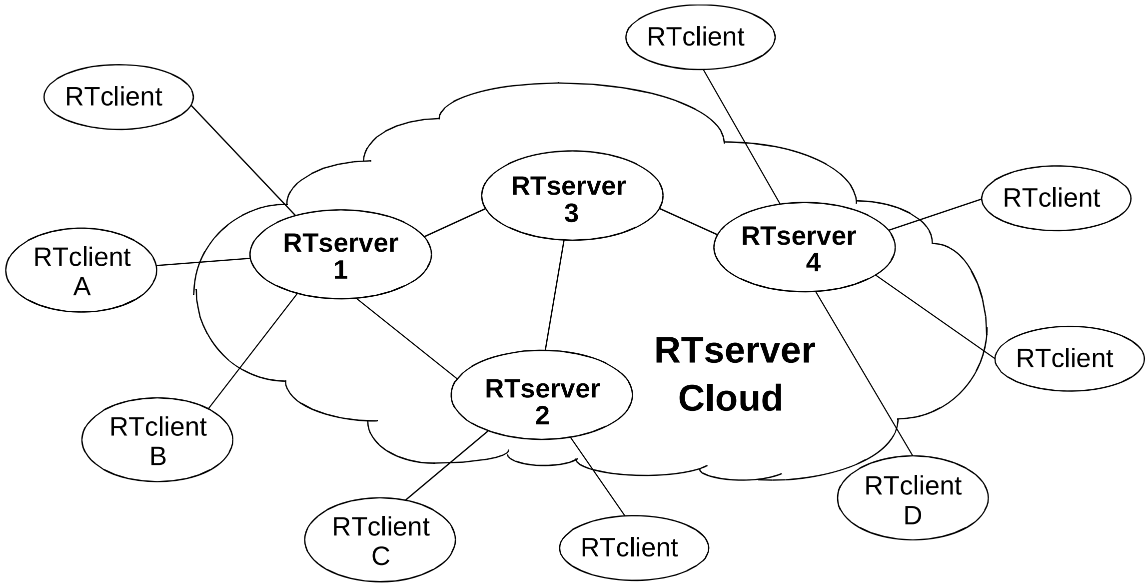
In addition to routing messages between RTclients, multiple RTservers can route messages to each other. Multiple RTservers can distribute the load of message routing. If an application is partitioned such that most of the messages being sent are routed between processes on the same node, then the use of multiple RTservers can reduce the consumption of network bandwidth (processes on the same node can use the non-network local IPC protocol). For more information, see the *TIBCO SmartSockets User's Guide*.

Connectivity

Each RTclient can have only one global connection, created using TipcSvc.getSrv, whether the connection is to an RTserver or to another process that plays a server-like role, such as RTgms. RTclients can connect to multiple RTservers if they use the TipcSvc.createSrv method, which creates connections independent of the global connection. Regardless of the type of connection, RTclients and RTservers do not have to be on the same network node. RTserver can run stand-alone, or it can connect to other RTservers. A message goes through one or more RTservers during delivery to an RTclient (or multiple RTclients). Messages are dynamically routed using a lowest cost algorithm, where each message passes through the fewest number of RTservers possible or, if paths have specified costs, the lowest cost path. (Lowest cost routing can be overridden, and message routing can be manually configured with RTserver subscribes.)

Figure 2 presents an example of the connectivity process in an RTserver cloud. In Figure 2, a message going from the RTclient A to the RTclient B goes through one RTserver. A message going from the RTclient A to the RTclient C goes through two RTservers. A message going from the RTclient A to the RTclient D goes through three RTservers. Routing is dynamic and can change at any time. Whenever a new RTserver becomes available or an existing RTserver goes down, routing tables in the RTservers are updated to reflect the new topology. For more information, see the *TIBCO SmartSockets User's Guide*.

Figure 2 Process Connectivity with RTserver Cloud



Running RTserver

RTserver is an independent SmartSockets process that can be run anywhere on the network. This chapter describes a few of the techniques for working with RTserver. More detailed information is presented in the *TIBCO SmartSockets User's Guide*.

Starting the RTserver

By default, the RTserver runs as a background process (on OpenVMS and Windows, this is known as a detached process). Because of security restrictions, Java RTclients cannot automatically start or restart RTservers, so it's important to make sure the RTserver is already running before your Java RTclient tries to connect. For details on starting the RTserver, see the *TIBCO SmartSockets User's Guide*.

On UNIX, you can start the RTserver manually to run as a background process with the `rtserver` command. Enter `rtserver` at the operating system prompt:

```
$ rtserver
```

On Windows, you can start the RTserver at the SmartSockets command prompt (`$`) or go the Start menu, select Programs, and select the SmartSockets program folder. Select RTserver.

On platforms that support both 32- and 64-bit, use the `rtserver64` command to run the 64-bit version of the `rtserver` script.

RTserver can be started automatically by C/C++ RTclients if it is not already running. As mentioned, the Java RTclients are restricted by the security settings of the Java Virtual Machine (VM) that executes them, and this means that Java RTclients cannot automatically start RTserver.

Stopping the RTserver

You can stop the RTserver with the `rtserver` command and its argument `-stop`. Execute `rtserver` on the computer where RTserver is running:

```
$ rtserver -stop
```

Add the argument `-server_names` *node* to stop the RTserver on a remote node. See the *TIBCO SmartSockets User's Guide* for more information.

Java RTclients cannot automatically restart the RTserver as the C/C++ RTclients can. Java RTclients enter an infinite loop, attempting to connect to RTserver, when the current connection is lost or an initial connection fails. This behavior continues until the connection is successfully completed, or the RTclient is stopped by the user or operating system.

RTserver Options

Options are available for both RTservers and RTclients. Options allow you to specify easily-modified parameters used by programs. For example, the RTserver Client_Max_Buffer option specifies the maximum number of message bytes that RTserver buffers for one of its RTclients. The RTclient option ss.user_name sets the name reported when a Java RTclient's owner information is requested.

Specific values for the RTserver options can be set in the `rtserver.cm` command file. Option values that have been specified in this command file are then set each time RTserver is started.

The Java RTclient options are documented in Chapter 9, RTclient Options. For more details about the RTserver options, see the *TIBCO SmartSockets User's Guide*.

What is a TIBCO SmartSockets Project?

A SmartSockets project is a group of RTclients working together with one or more RTservers to achieve the goals of a specific system. Within a project, RTclients and RTservers can communicate with other RTclients and RTservers on the same machine or over the network. However, RTclients in different projects cannot send messages to each other.

An RTclient can belong to only one project at a time. An RTserver process does not belong to any specific project, but it can provide message routing services for one or more projects simultaneously. A project can be thought of as a firewall that prevents messages from being dispatched outside the specified RTclient group.

A project is designated by a name, which must be an identifier (often the application's name is used for the project name). The default project name is `rtworks`. You can change the default project name using the ss.project option. You should set this option to prevent your Java RTclients from becoming part of the default `rtworks` project; otherwise, unwanted messages may be received. Remember that in the Java library, all of the standard SmartSockets options are prefixed with ss.; in SmartSockets C programs, the equivalent option is simply `project`.

In the sending and receiving programs you wrote in Lesson 1, the ss.project option was not explicitly set. This resulted in these programs being part of the `rtworks` default project. You should set this option to build a firewall between your application and other SmartSockets applications. This example shows you how to change the project name:

The files for this lesson are located in the directories:

Windows:

`%RTHOME%\java\tutorial\lesson2`

UNIX:

`$RTHOME/java/tutorial/lesson2`

Step 1 **Modify the sending program**

Modify the sending program from the previous lesson to look like this example, or copy the `send.java` file into your working directory:

```
//-------------------------------------------------------
// send.java
1 import java.io.*;
2 import com.smartsockets.*;

3 public class send {
4   public static void main(String[] argv) {
      try {
5       Tut.setOption("ss.project", "smartsockets");
6       TipcSrv srv=TipcSvc.getSrv();
7       if (!srv.create()) {
8         Tut.exitFailure("Couldn't connect to RTserver!");
        }
9       TipcMsg msg = TipcSvc.createMsg(TipcMt.INFO);
10      msg.setDest("/ss/tutorial/lesson2");
11      msg.appendStr("Hello World!");
12      srv.send(msg);
13      srv.flush();
14      srv.destroy();
15    } catch (TipcException e) {
16      Tut.warning(e);
      } // catch
     } // main
  }  // send
```

Let’s examine some of the key lines in your new sending program:

Line 5 Sets the project name to `smartsockets`. Now only processes that belong to the `smartsockets` project can communicate with your sending program. Note that all the members of the `Tut` class are static, so an instance of the `Tut` class need not (and should not) be created.

Line 7 Explicitly tries to make a connection to RTserver using the TipcSrv create() method. Be sure this method is used within a `try` block, so that the IOException that is potentially thrown can be handled. Also, check the return value as shown below, because a false result indicates a connection could not be made.

```
if (!srv.create()) {
  Tut.exitFailure(
    "Couldn't connect to an RTserver!");
}
```

In this example, the program exits if a connection cannot be made. Another alternative is to try connecting to an RTserver process on another network node.

Line 12 Publishes the INFO message to the `/ss/tutorial/lesson2` subject.

Line 14 Explicitly disconnects from RTserver, ensuring the data is flushed and completes our connect-publish-disconnect cycle.

Compile and run the sending and receiving programs, as was done in the previous lesson. (If RTserver is not still running, start it now.)

Step 2 **Compile the sending program**

Compile the modified `send.java` program:

```
$ javac send.java
```

Step 3 **Start the receiving program first**

Start the receiving and sending programs in separate windows, as you did in Lesson 1. First start the receiving program (the same one used in Lesson 1):

```
$ java receive
```

Step 4 **Start the sending program**

After a few moments, start the sending program:

```
$ java send
```

Step 5 **Change the project and subscribe to the correct subject**

Notice that the receiving program did not read nor print the message from the sending program. This is because you set the ss.project option in the sending program to `smartsockets` and have not yet set the ss.project option in the receiving program. The receiving program still belongs to the default `rtworks` project. RTserver prevents the message sent by the sending program from being delivered to the receiving program, because it is in a separate project. In addition, the receiving program is still subscribing to the `/tutorial/lesson2` subject.

To fix these problems, modify `receive.java` to belong to the same project as `send.java` and to subscribe to the `/tutorial/lesson2` subject.

Step 6 Modify the receiving program

Modify the receiving program from the previous lesson to match this example, or copy the `receive.java` file (from the `lesson2` directory) into your working directory:

```
//----------------------------------------------------------
// Program 2: receive.java

import java.io.*;
import com.smartsockets.*;

public class receive {

  public static void main(String[] argv) {
    TipcMsg msg = null;
    String text = null;

    try {
      Tut.setOption("ss.project", "smartsockets");
      TipcSrv srv=TipcSvc.getSrv();
      if (!srv.create()) {
         Tut.exitFailure("Couldn't connect to RTserver!");
      }
      srv.setSubjectSubscribe("/ss/tutorial/lesson2", true);
      msg = srv.next(TipcDefs.TIMEOUT_FOREVER);

      msg.setCurrent(0);
      text = msg.nextStr();
    } catch (TipcException e) {
    Tut.fatal(e);
    } // catch

  System.out.println("Text from INFO message = " + text);
  } // main
} // receive
```

Now you need to compile the modified receiving program, and then you can run it with the sending program you ran earlier.

Step 7 Compile the receiving program

Compile the receiving program:

```
$ javac receive.java
```

Step 8 **Start the receiving program first**

Start the receiving and sending programs in separate windows as you did earlier in the lesson. Make sure RTserver is running. Start the receiving program:

```
$ java receive
```

Step 9 **Start the sending program**

After a few moments, start the sending program:

```
$ java send
```

This output is displayed by the receiving program:

```
Text from INFO message = Hello World!
```

The modified sending and receiving programs communicate using the `/ss/tutorial/lesson2` subject in project `smartsockets`.

What are Subjects?

Just as projects restrict the boundaries of where messages are sent, subjects partition the flow of messages within 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 publish a message to multiple processes with a single operation. Subjects are designated by a name, which can be any character string with a few restrictions.

A message in SmartSockets has both a sender and a destination property. (See the *TIBCO SmartSockets User's Guide* for a full discussion of message properties.) When TipcConn peer-to-peer methods are used to send messages through connections, the sender and destination properties are not used. There are no predefined values for these properties when working with peer-to-peer connections.

For RTserver to RTclient communication, however, subjects specify the sender and destination properties. When an RTclient subscribes to a subject, it receives any published messages whose destination property is set to that subject. Think of this as the process signing up for messages sent to a particular subject. For example, in a network monitoring application, you might partition messages by the types of items to be monitored—routers, bridges, switches, and so on. These areas can be declared as subjects such as `/router`, `/bridge`, and `/switch`. All messages pertaining to routers are constructed with the `/router` subject as their destination property. Any RTclient interested in receiving messages about routers subscribes to the `/router` subject. This is also known as the publish-subscribe paradigm because RTclients publish messages to specific subjects and subscribe to subjects in which they are interested.

The SmartSockets publish-subscribe communications model allows an RTclient to effortlessly publish messages to multiple receivers. Simply by specifying a subject in the `destination` property, you ensure that RTserver routes the message to all other RTclients in the same project that are subscribed to that subject. The RTclients can subscribe to or unsubscribe from subjects at any time, which allows the RTclients to control the quantity of incoming messages.

Understanding Hierarchical Subject Namespace

To provide greater flexibility and scalability for large applications, SmartSockets subject names are arranged in a hierarchical namespace, much like UNIX file names or World Wide Web URLs. This hierarchical namespace allows a large numbers of subject names to be created with similar, but not conflicting, names; for example, `/stocks/NYSE/computer` and `/stocks/NASDAQ/gold`. Many powerful operations, such as publish-subscribe with wildcards, can be performed in this namespace model. Small SmartSockets systems can be easily built without requiring large amounts of complexity, and large systems 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 slashes (/). Each component can contain any characters except a slash, an asterisk (*), or an ellipsis (...), the latter two of which are used for publish-subscribe wildcards. This list represents some examples of hierarchical subject names:

- `/system, /system/primary/eps`
- `/system/backup/eps`
- `/nodes/workstation1.talarian.com/ssuser`

The hierarchy can be specified to any depth. For more details on hierarchical subject names, see the *TIBCO SmartSockets User's Guide*.

Specifying Wildcards in Subjects

When subscribing or publishing to a subject, you can use wildcards in the specification of the subject name to match multiple subjects. Using wildcards in subjects is much like using wildcards for file names on an operating system command line. The asterisk (`*`) wildcard operates much the same as it does on Windows, UNIX, or OpenVMS. 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 across components, for example, `"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, for example, `auto...` is not a wildcard. Multiple wildcards can be combined in a single subject name, for example, `/a*b*/.../d`. For more details on using wildcards with subjects see the *TIBCO SmartSockets User's Guide*.

Demonstrating Message Routing

This section illustrates how a message originating from a single RTclient is published to multiple RTclients subscribed to the specified subject. In Figure 3 on page 43, processes are represented by circles, and connections between processes are represented by dark lines. As you can see, there is a single sending process (Send), and two receivers (Receive1 and Receive2). Each of these RTclients is connected to the same RTserver. The Receive1 and Receive2 programs have both subscribed to the `/sub1` subject. If the Send process wants to publish a message to the `/sub1` subject, this sequence of events occurs:

1. The sending program constructs a message with `/sub1` as the destination subject.
2. The sending program publishes the message.
3. RTserver receives the message.
4. RTserver looks at the destination (`/sub1`) of the message.
5. RTserver publishes the message to all RTclients currently subscribed to the `/sub1` subject.
6. Both Receive1 and Receive2 receive a copy of the message.

Figure 3   RTserver Message Routing

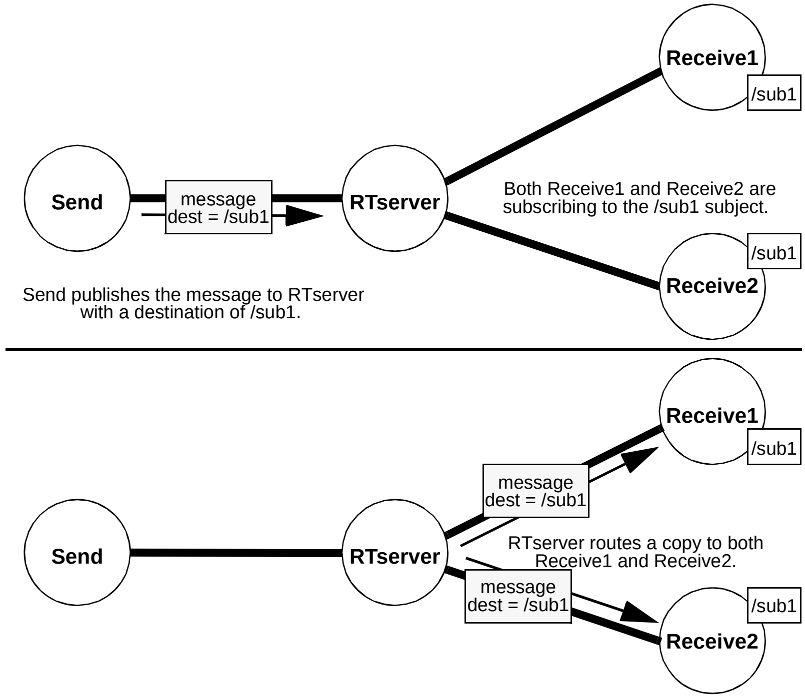


Figure 3 shows the message flow through RTserver. Note that if the Send program was also subscribed to the `/sub1` subject, it too would receive a copy of the message from RTserver.

Demonstrating Publish-Subscribe Services

Let's go back to the sending and receiving examples earlier in this lesson. Line 10 of the sending program calls `setDest`, as shown.

```
msg.setDest("/ss/tutorial/lesson2");
```

The `setDest` method specifies the subject the message is being published to. For the receiving program to get the message, line 11 of the receiving program is required:

```
srv.setSubjectSubscribe("/ss/tutorial/lesson2", true);
```

This line allows the receiving program to start receiving any messages published to the `/ss/tutorial/lesson2` subject. Note the second parameter is set to `true`. If this had been set to `false`, it would indicate that the receiving program is not going to receive messages published to the `/ss/tutorial/lesson2` subject. Setting the second parameter to `false` unsubscribes the process from the specified subject.

Step 10 **Start the receiving program again**

To illustrate this concept more clearly, you need three separate windows open. In the first window, make sure RTserver is running, then start up the receiving program:

```
$ java receive
```

Step 11 **Edit the receiving program**

Now edit the receiving program, changing line 11 so the receiving program subscribes to the `/smartsockets/foo` subject. Line 11 should now look like:

```
srv.setSubjectSubscribe("/smartsockets/foo", true);
```

Step 12 **Save the modified receiving program and compile**

Save your changes to `receive.java`. Compile the program again:

```
$ javac receive.java
```

Step 13 **Start the new receiving program**

In the second window, start up the new receiving program:

```
$ java receive
```

Step 14 **Start the sending program**

Finally, in the third window, start up the sending program:

```
$ java send
```

You should see that your original receiving program got the message and printed it.

```
Text from INFO message = Hello World!
```

Notice that the new receiving program (subscribed to the `/smartsockets/foo` subject) did not receive the message. In fact, it is still blocked, waiting for a message. Stop the blocked program's execution with Ctrl-c and return to the operating system prompt.

Step 15 **Edit the receiving program again to subscribe to another subject**

To see how one message can be delivered to two processes in a single operation, go back and edit the receiving program to once again receive the `/ss/tutorial/lesson2` subject. Line 11 should now look like:

```
srv.setSubjectSubscribe("/ss/tutorial/lesson2", true);
```

Step 16 **Compile the modified receiving program**

Compile the program as before:

```
$ javac receive.java
```

Step 17 **Start the new receiving program**

Now start the new receiving program in the first window:

```
$ java receive
```

Step 18 **Start a second instance of the same receiving program**

In the second window, start up a second instance of the same receiving program:

```
$ java receive
```

Step 19 **Start the sending program**

Finally, in the third window, start up the sending program:

```
$ java send
```

In a few moments, you should see that both windows where the receiving programs are running display:

```
Text from INFO message = Hello World!
```

If you wish, you can start any number of receiving programs, run the sending program, and observe as the message gets delivered to all of them with a single operation.

Note that you did not have to change a single line of code in the sending program to take advantage of this desirable feature. The ability to send a message to multiple processes with a single operation, without having to specify the location of the processes, is a key feature of SmartSockets. In addition to providing useful functionality, this feature makes the testing, debugging, and maintenance of your network application much easier. Through the use of subjects and SmartSockets publish-subscribe services, you also achieve location transparency. This implies that your programs can be easily relocated anywhere on your network without changing a single line of code.

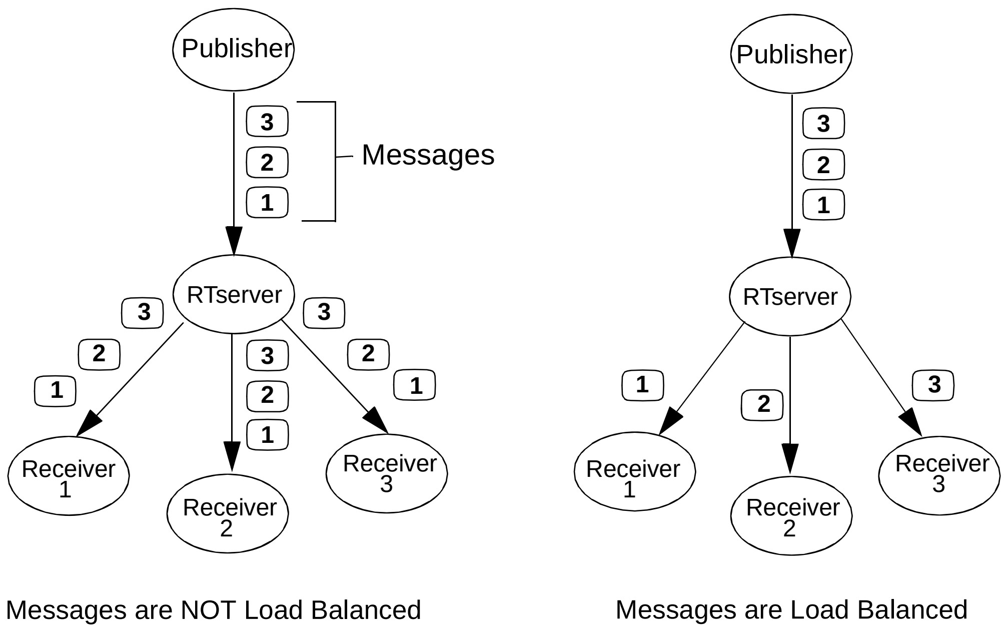
Using Load Balancing

In normal publish-subscribe operations, a message is published to all RTclients subscribing to the subject to which the message is being sent. However, in some situations you may wish to have messages sent to only one of a specified set of RTclients. 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 (or have the least busy one handle) the messages to better keep up with message flow.

This is accomplished in SmartSockets with 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 4. 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 4 Messages Delivered With and Without Load Balancing



Load balancing can be specified on a per-message basis or per-message-type basis through the load balancing mode message property. Load balancing is dynamic in that whenever an RTclient connects to or disconnects from 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 other RTservers to accurately track load balancing accounting. This increases the scalability of load balancing because only the relevant RTservers dynamically exchange load balancing information.

By default, messages are not load balanced and are distributed to all subscribers. Setting the load balance mode per message takes precedence over message type. Setting load balancing for a message type takes precedence over the default value of `TipcDefs.LB_NONE`.

SmartSockets supports four load balancing modes:

TipcDefs.LB_NONE	is the default and specifies no load balancing. The message is sent to all subscribers.
TipcDefs.LB_ROUND_ROBIN	specifies that the list of subscribing RTclients is held in a circular list, with each successive message simply sent to the next RTclient in the list (as shown in Figure 4). 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.
TipcDefs.LB_WEIGHTED	specifies that the message is published to the RTclient that has the fewest pending requests. This mode is a good choice when the subscribers differ significantly in their ability to process a request promptly, due to, for example, hardware speed or network delays. This method can only be used with GMD and requires no additional overhead beyond what GMD requires.
TipcDefs.LB_SORTED	specifies that 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. 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.

For further information on load balancing, refer to the *TIBCO SmartSockets User's Guide*.

Connecting to RTserver on Another Node

Up to this point, all of our sample programs have connected to an RTserver process running on the same node. This is the default behavior of SmartSockets. If you do not specify where the RTclient should try to find and connect to RTserver, it always looks locally first.

This default behavior can be changed by setting the ss.server_names option for the Java RTclient. This option can be used to specify where RTserver is located, as well as what protocol to use when connecting. The ss.server_names option contains a list of machines with which the RTclients attempt to establish a connection.

Just as with the ss.project option, the ss.server_names option is set programmatically. For more details on how an RTclient finds RTserver and connects to it, see the *TIBCO SmartSockets User's Guide* for more information on starting the RTserver. Keep in mind, however, that Java clients can only connect to running RTservers; Java clients cannot start new RTservers. For more details on setting options (including ss.project,) see Lesson 5: TIBCO SmartSockets Options.

Disconnecting from RTserver

Every RTclient program should call the TipcSrv.destroy method before exiting. An RTclient using the TCP/IP protocol to connect to RTserver may lose outgoing messages if the process terminates without calling TipcSrv.destroy. The TipcSrv.destroy method forces the operating system to deliver all outgoing messages. Invoking `flush` is not, in itself, enough to guarantee message delivery immediately prior to a program's termination.

When the connection to the RTserver is destroyed by calling TipcSrv.destroy(`TipcSrv.CONN_NONE`) all server create and server destroy callbacks are also destroyed.

If an RTclient has registered any server create or server destroy callbacks then these callbacks are destroyed when TipcSrv.destroy(`TipcSrv.CONN_NONE`) is called.

Summary

The key concepts covered in this lesson are:

- SmartSockets provides both peer-to-peer and client-server communications models (with the TipcConn and TipcSrv classes, respectively).
- RTserver is a standard SmartSockets process used to implement the publish-subscribe communications services, allowing location transparency of RTclients.
- An RTclient connects to a single RTserver. An RTserver process can connect to other RTservers. Messages are dynamically routed from an RTclient to other RTclients through one or more RTservers using a lowest cost algorithm.
- RTserver is not started automatically by Java RTclients if it is not already running. RTserver can be started manually using the `rtserver` shell script. RTserver can be stopped manually using the `rtserver` shell script with the `-stop` command line argument.

On platforms that support both 32- and 64-bit, use the `rtserver64` command to run the 64-bit version of the `rtserver` script.

- RTserver runs as a background (detached) process.
- By default, a client program tries to connect to an RTserver process on its same node. A client program can connect to an RTserver on another node by setting the ss.server_names option.
- The ss.project option is used by RTclients to prevent processes in another project from communicating with them.
- Subjects are logical addresses set as the destination property of a message. Subjects are used by RTserver to dynamically route messages to all RTclients subscribed to that subject. This allows a single message to be delivered to multiple RTclients with a single operation.
- Subjects are the fundamental unit used by SmartSockets to implement publish-subscribe services.
- Subject names can be specified in a hierarchical manner, and to any number of levels, for example, `/company/software/tibco`.
- Rather than deliver a message to all RTclients that have subscribed to a subject, a message can be delivered to only one of a set of RTclients using load balancing.
- Load balancing can be set on a per-message or per-message-type basis.

- Wildcards, asterisks (*) and ellipsis (...), can be used when subscribing or publishing to a subject.
- Always be sure to call the TipcSrv's destroy method before your program exits to disconnect from RTserver and flush any messages still in the buffer.

Chapter 5 Lesson 3: Messages

In this lesson you learn about:

- what a message is
- how to construct and send messages
- how to receive and access messages
- how to use the SmartSockets Java Class Library to operate on messages

Topics

Lesson 3 Overview

The files for this lesson are located in the directories:

Windows:

`%RTHOME%\java\tutorial\lesson3`

UNIX:

`$RTHOME/java/tutorial/lesson3`

As seen in the previous lessons, within a SmartSockets application, interprocess communication occurs through messages. A message is a structured packet of information sent from one process to one or more other processes providing instructions or data. Messages can carry many different kinds of information in a SmartSockets application, including: alarms, variable-value pairs representing sensor information, and IPC information about a client.

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

What is in a Message?

A message is composed of several parts, or properties. The most important property is the message data. All parts of a message can be accessed directly with the SmartSockets API. Almost all parts of a message can be specified using the API. With the exception of the message create method, which belongs to the TipcSvc factory class, all methods for working with messages are members of the TipcMsg class. Messages can be created with TipcSvc.createMsg and copied with TipcMsg.clone.

Figure 5 shows an example of the message that you constructed and sent in the previous lesson (Lesson 2: Publish-Subscribe). This message was a standard SmartSockets message type, `INFO`, and was sent to the `/ss/tutorial/lesson2` subject. The data part of the message consisted of a single string field containing the string "`Hello World!`".

Figure 5   Composition of a Typical Message

Message Composition			
Type		INFO	
Sender		/_workstation1_5415	
Destination		/ss/tutorial/lesson2	
Priority		0	
Delivery Mode		T_IPC_DELIVERY_NONE	
Delivery Timeout		0.0	
Load Balancing Mode		T_IPC_LB_NONE	
Header String Encode		FALSE	
Reference Count		1	
Sequence Number		0	
User-Defined Property		0	
Read Only		FALSE	
Data	Field	Type	str
		Value	Hello World!

There are SmartSockets methods to get (access) and set (write) each part of the message.

A message is composed of several properties. The SmartSockets Java Class Library methods for setting and getting the message property are enclosed in parentheses.

Type
: is the kind of message being manipulated (TipcMsg.setType, TipcMsg.getType).

Sender
: is the name of the originator of a message (TipcMsg.setSender, TipcMsg.getSender). SmartSockets automatically fills this in for you when using RTserver to deliver the message.

Destination
: is the name of the subject where a message is going (TipcMsg.setDest, TipcMsg.getDest).

Priority
: is the level of importance of a message (TipcMsg.setPriority, TipcMsg.getPriority).

Delivery Mode
: is the level of guarantee when a message is sent through a connection (TipcMsg.setDeliveryMode, TipcMsg.getDeliveryMode).

Delivery Timeout	is the number of seconds specifying how long to wait for acknowledgment of delivery of a message sent through a connection (TipcMsg.setDeliveryTimeout, TipcMsg.getDeliveryTimeout).
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 (TipcMsg.setLbMode, TipcMsg.getLbMode).
Header String Encode	controls whether or not header strings are converted to four-byte integers when a message is sent through a connection. This field cannot be directly accessed by Java clients.
Reference Count	is the number of independent references to a message. This field cannot be directly accessed by Java clients.
Sequence Number	uniquely identifies a message for guaranteed message delivery (TipcMsg.getSeqNum). SmartSockets assigns this number and it cannot be manually set.
User-defined Property	is a user-defined value that can be used for any purpose (TipcMsg.setUserProp, TipcMsg.getUserProp). This field is not used internally by SmartSockets.
Read Only	controls whether or not a message can be modified (TipcMsg.getReadOnly). SmartSockets automatically sets this property and it cannot be manually set.
Data	is the instructions or value part of a message (TipcMsg.append*, TipcMsg.next*).

There are a large number of SmartSockets methods to build the data part of a message (TipcMsg.append*) and access the data part of a message (TipcMsg.next*). See the online Java documentation for a complete description of these methods.

Figure 6 shows an example of a more complex message. It is a SmartSockets standard NUMERIC_DATA message, and the data part of this message is a series of variable name-value pairs (`voltage = 33.4534`, `switch_pos = 0`). You construct a message similar to this one later in this lesson.

Typically, the data part of the message is the largest part of the message.

Figure 6 Composition of a NUMERIC_DATA Message

<table>
<tr><th colspan="4">Message Composition</th></tr>
<tr><td colspan="2">Type</td><td colspan="2">NUMERIC_DATA</td></tr>
<tr><td colspan="2">Sender</td><td colspan="2">/_workstation1_5415</td></tr>
<tr><td colspan="2">Destination</td><td colspan="2">/system/thermal</td></tr>
<tr><td colspan="2">Priority</td><td colspan="2">10</td></tr>
<tr><td colspan="2">Delivery Mode</td><td colspan="2">TipcDefs.DELIVERY_ALL</td></tr>
<tr><td colspan="2">Delivery Timeout</td><td colspan="2">20.0</td></tr>
<tr><td colspan="2">Load Balancing Mode</td><td colspan="2">TipcDefs.LB_WEIGHTED</td></tr>
<tr><td colspan="2">Header String Encode</td><td colspan="2">TRUE</td></tr>
<tr><td colspan="2">Reference Count</td><td colspan="2">1</td></tr>
<tr><td colspan="2">Sequence Number</td><td colspan="2">3892675</td></tr>
<tr><td colspan="2">User-Defined Property</td><td colspan="2">42</td></tr>
<tr><td colspan="2">Read Only</td><td colspan="2">FALSE</td></tr>
<tr><td rowspan="8">Data</td><td rowspan="2">Field</td><td>Type</td><td>str</td></tr>
<tr><td>Value</td><td>voltage</td></tr>
<tr><td rowspan="2">Field</td><td>Type</td><td>real8</td></tr>
<tr><td>Value</td><td>33.4534</td></tr>
<tr><td rowspan="2">Field</td><td>Type</td><td>str</td></tr>
<tr><td>Value</td><td>switch_pos</td></tr>
<tr><td rowspan="2">Field</td><td>Type</td><td>real8</td></tr>
<tr><td>Value</td><td>0.0</td></tr>
</table>

What is Automatic Data Translation?

One of the key features of SmartSockets is that it has structured messages. There is no need for you to figure out how to encode your messages. SmartSockets takes care of that for you and provides robust methods that allow you access to any part of the structure of a message. Because the messages are structured, SmartSockets knows how to automatically convert different data types when delivering the message across a heterogeneous network. This is all done transparently for you, using a receiver-makes-right approach in which the final receiver of the message does the translation. This is most efficient, as data translation is only done once.

Many other messaging products do not have the concept of a structured message type. They simply move a block of memory across the network. There is no API to help build and access the different fields of the message, and there is no automatic conversion of data. They leave these tasks up to you, increasing the amount of time it takes to build your application.

What are Message Types?

As described earlier, each message has a type property that defines the structure of the data property of a message. A message type can be thought of as a template 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 is then available for use as the type for any number of messages.

SmartSockets provides dozens of standard message types that cover a wide variety of different types of information that can be passed. SmartSockets standard message types allow you to begin building your application quickly, without having to figure out how to define your own message types. When there is no standard message type to satisfy your specific need, you can easily create a user-defined message type. Both standard and user-defined message types are handled in the same manner and can co-exist within the same program and application. Once the message type is created, messages can be constructed, sent, received, and processed through a variety of methods.

Table 1 lists some of the frequently-used 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 the *TIBCO SmartSockets User's Guide*.

Table 1 Standard Message Types

Message Type (Tipc.Mt)	Grammar	Description
ALERT	`id` */*object*/* `str` */*message*/*	alert message
BOOLEAN_DATA	`{ id` */*name*/* `bool` */*value*/* `}`	boolean data values
CANCEL_ALERT	`id` */*object*/* `str` */*message*/*	cancel an alert
CANCEL_WARNING	`id` */*object*/* `str` */*message*/*	cancel a warning
CONN_INIT	`str` */*unique_subject*/* `id` */*node*/* `id` */*user*/* `int4` */*pid*/* `id` */*arch*/*	one-time connection initialization information exchanged when RTclient and RTserver rendezvous
CONNECT_CALL	`id` */*project*/* `str` */*ident*/* `int2` */*disconnect_mode*/* `str_array` */*init_subscribes*/* `int4_array` */*lb_status*/*	information RTclient supplies when connecting to RTserver
CONNECT_RESULT	`bool` */*status_flag*/* `str` */*status_output*/* `str` */*def_subj_prefix*/*	information RTserver supplies back to RTclient

Table 1 Standard Message Types

Message Type (Tipc.Mt)	Grammar	Description
DISCONNECT	`int2` */*disconnect_mode*/*	RTclient explicitly disconnecting from RTserver
ENUM_DATA	`{ id` */*name*/* `id` */*value*/* `}`	enumerated data values
INFO	`str` */*message */*	information message
NUMERIC_DATA	`{ id` */*name*/* `real8` */*value*/* `}`	numeric data values
SERVER_STOP_CALL	`int` */*stop_type*/*	request RTserver to shut itself down
SERVER_STOP_RESULT	`str` */*result_output*/*	information RTserver supplies as it is shutting down
STRING_DATA	`{ id` */*name*/* `str` */*value*/* `}`	string data values
SUBJECT_SET_SUBSCRIBE	`id` */*subject*/* `bool` */*subscribe_flag*/* `bool` */*lb_flag*/*	start or stop subscribing to a subject
WARNING	`id` */*object*/* `str` */*message*/*	warning message

Any message type can be looked up, either by name or numeric ID, with the overloaded method TipcSvc.lookupMt. For example, these two lines are equally effective:

```
mt = TipcSvc.lookupMt("numeric_data");
mt = TipcSvc.lookupMt(TipcMt.NUMERIC_DATA);
```

Working With Messages

Typically, these three steps are required when constructing a message:

1. Create a message of a particular type (TipcSvc.createMsg, TipcMsg.clone)
2. Set the properties of the message (TipcMsg.set*)
3. Append fields to the message data (TipcMsg.append*)

Many different types of fields can be appended to a message's data. These field types include three sizes of integers, two sizes of real numbers, character strings, and arrays of the scalar field types (such as an array of four-byte integers). These field types are listed in the online documentation for the TipcMsg class and can be recognized by their `FT_` prefix. Messages themselves can even be used as fields within other container messages; this allows operations such as large transactions to be represented with a single message. Once a message is constructed, it can be published to other RTclients using the TipcSrv.send method.

To get a better feel for working with the SmartSockets Java API for building and sending messages, you will enhance your sending program from the previous lesson to send a NUMERIC_DATA message. The data part of a NUMERIC_DATA message consists of one or more variable-value pairs, where a variable is a text string and a value is a floating point number.

The files for this lesson are located in the directories:

Windows:

```
%RTHOME%\java\tutorial\lesson3
```

UNIX:

```
$RTHOME/java/tutorial/lesson3
```

Step 1 **Copy the send2.java program**

Copy the file `send2.java` into your working directory:

```
//-----------------------------------------------------------
// send2.java -- write a NUMERIC_DATA message

import java.io.*;
import com.smartsockets.*;

public class send2 {

  public static void main(String[] argv) {
    try {
      Tut.setOption("ss.project", "smartsockets");
```

```
6        TipcSrv srv=TipcSvc.getSrv();
7        if (!srv.create()) {
8          Tut.exitFailure("Couldn't connect to RTserver!");
         }
     // create a message of type NUMERIC_DATA
9        TipcMsg msg = TipcSvc.createMsg(TipcMt.NUMERIC_DATA);

     // set the destination subject of the message
10       msg.setDest("/ss/tutorial/lesson3");

     // build a NUMERIC_DATA msg with 3 variable-value pairs,
     // set as follows (X, 10.0), (Y, 20.0) and (Z, 30.0)
11       msg.appendStr("X");
12       msg.appendReal8(10.0);
13       msg.appendStr("Y");
14       msg.appendReal8(20.0);
15       msg.appendStr("Z");
16       msg.appendReal8(30.0);

     // send the message
17       srv.send(msg);
18       srv.flush();

     // disconnect from RTserver
19       srv.destroy();
       } catch (TipcException e) {
20       Tut.warning(e);
       } // catch
     } // main
   } // send2
```

Let's examine some of the key lines in this program:

Line 9 — Now creates a message of type NUMERIC_DATA instead of type INFO.

Lines 11-16 — The call to TipcMsg.appendStr is replaced with multiple calls to TipcMsg.appendStr and TipcMsg.appendReal8 to build the data part of the message.

Step 2 Compile the send2.java program

Now compile the `send2.java` program.

```
$ javac send2.java
```

You now need to modify your receiving program so that it can read and print the contents of the NUMERIC_DATA message you are sending.

Step 3 **Copy the receive2.java program**

Copy the file `receive2.java` into your working directory. This is an example of the `receive2.java` program:

```
//-------------------------------------------------------------
// receive2.java -- receive a NUMERIC_DATA message

import java.io.*;
import com.smartsockets.*;

public class receive2 {

  public static void main(String[] argv) {
    TipcMsg msg = null;
    TipcSrv srv = null;

  // set the ss.project
    try {
      Tut.setOption("ss.project", "smartsockets");
      srv=TipcSvc.getSrv();

   // connect to RTserver
      if (!srv.create()) {
        Tut.exitFailure("Couldn't connect to RTserver!");
      } // if
   // subscribe to the appropriate subject
      srv.setSubjectSubscribe("/ss/tutorial/lesson3", true);
      msg = srv.next(TipcDefs.TIMEOUT_FOREVER);
    }
      catch (TipcException e) {
      Tut.fatal(e);
    } // catch

  // position the field ptr to the beginning of the message
    try {
      msg.setCurrent(0);
   } catch (TipcException e) {
     Tut.fatal(e);
   } // catch

   System.out.println("Contents of NUMERIC_DATA message:");

  // read the data part of the message
   try {
     String var_name;
     while (true) {
       var_name = msg.nextStr();
       double var_value;
       var_value = msg.nextReal8();
       System.out.println("Var name = " + var_name +
               ", value = " + var_value);
      } // while
    }
```

```
       // catch end-of-message-data exception, do nothing.
25         catch (TipcException e) {
        // we expect this exception from the while loop
           } // catch

       // drop our connection to RTserver
           try {
26           srv.destroy();
27         } catch (TipcException e) {
             // not concerned with problems dropping connection
           } // catch
         } // main

       } // receive2
```

Let's look at how this program extracts information from the data part of the message:

Lines 7-14 Set the project to "`smartsockets`" and connect to RTserver.

Line 20 Beginning of loop over fields in the data part of the message.

Lines 21-23 The TipcMsg.nextStr method retrieves the variable name and the TipcMsg.nextReal8 method retrieves the variable value. When TipcMsg.nextStr throws a NoSuchFieldException, there are no more fields to access in the message, and the enclosing while loop is exited.

This is an expensive way to end the loop (throwing exceptions takes a lot of time), and you can also use the TipcMsg.getNumFields method to retrieve the field count and loop with a counter instead.

For more details, see the descriptions of the TipcMsg.next* methods in the online Java documentation.

Step 4 **Compile the receive2.java program**

Now compile the `receive2.java` program using the command:

```
$ javac receive2.java
```

Step 5 **Start the receiving program**

To demonstrate that everything is still working, start the receiving and sending programs in separate windows as you did earlier. First start the `receive2` program using the command:

```
$ java receive2
```

Step 6 **Start the sending program**

After a few moments, start the sending program:

```
$ java send2
```

This message output is displayed when you run the receiving program:

```
Contents of NUMERIC_DATA message:
---------------------------------
Var name = X, Var Value = 10
Var name = Y, Var Value = 20
Var name = Z, Var Value = 30
```

Named Fields

Not only can you add and access fields to a message sequentially as demonstrated in the previous example, you can also add, access, update, and delete fields in messages by name. A name is associated with a field when the field is added to the message, using the TipcMsg.addNamed*Type* methods. Once you have added a named field to a message, you can:

- access it using the TipcMsg.getNamed*Type* methods
- update it using the TipcMsg.updateNamed*Type* methods
- delete it using the TipcMsg.deleteNamedField method

Named fields and those without names can co-exist in the same message without conflict. In addition, named fields can be accessed either using their name or sequentially, like any other field.

This example shows how to add a named field and how to access it both by name and sequentially:

```
import com.smartsockets.*;

/**
 * Named fields example program.
 */

public class NamedFieldsExample {

  public static void main(String[] argv) {

    try {
      /* Create a message */
      TipcMsg msg = TipcSvc.createMsg(TipcMt.INFO);

      /* Add a non-named int4 field, and a named string field */
      msg.appendInt4(5);
      msg.addNamedStr("string one", "hello");
```

```
      /* Now get the string field */
      String str = msg.getNamedStr("string one");
      System.out.println("named string field is " + str);

      /* Rewind the index back to the first field, and get the int4 field */
      msg.setCurrent(0);
      int i = msg.nextInt4();
      System.out.println("first field is " + 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.
    */
      str = msg.nextStr();
      System.out.println("second field is " + str);

      /*
    * Rewind the index pointer again, and we can "name" the int4 field.
    */
      msg.setCurrent(0);
      msg.setNameCurrent("int4 zero");

      /*
    * We can also get the name of the current field.
    */
      str = msg.getNameCurrent();
      System.out.println("name of first field is " + str);
    }
    catch (Exception e) {
      Tut.fatal(e);
    }
  }
}
```

Summary

The key concepts covered in this lesson are:

- Message types are structured templates that describe what can go in a message, and each message can be considered an instance of a message type.
- SmartSockets comes with a number of ready-to-use standard message types. You can also define your own message types.
- A message consists of a set of header properties and a data part. The properties in a message are the same across all message types. Properties include its type, sender, destination, priority, read-only status, delivery mode, delivery timeout, load balancing mode, header string encode, reference count, sequence number, and a user-definable property.
- The data part of a message consists of fields that carry a unit of information. The message data can contain any number of fields, although most messages have a well-defined layout for their fields.
- SmartSockets converts all data to the proper format using a final-receiver-makes-right approach. You do not have to worry about any data translation between different platforms, except for IEEE to DEC floating-point conversion, which is currently not available in the SmartSockets Java Class Library.
- The SmartSockets Java Class Library has an extensive set of functions that allows you to create and copy messages, and to get and set any of the message properties, and to build and access the data part of a message.
- Typically, these steps are required when constructing a message:
    a. Create a message of a particular type (TipcSvc.msgCreate, TipcMsg.clone).
    b. Set the properties of the message (TipcMsg.set*).
    c. Append fields to the message data (TipcMsg.append*).
- Messages are read from RTserver using TipcSrv.next.
- Typically, retrieving information from the data part of a message consists of these steps:
    a. Set the message pointer to the field of interest (TipcMsg.setCurrent).
    b. Access the fields of the message (TipcMsg.next*).
- Individual fields in messages can be associated with a name. Named fields can be accessed by name or sequentially. Named fields and non-named fields can co-exist in the same message without conflict.

Chapter 6 Lesson 4: Callbacks

In this lesson you learn about:

- what callbacks are
- what types of callbacks are available in SmartSockets
- how to write a process and a default callback
- how to use server create and server destroy callbacks
- how to write a subject callback
- how to read and process multiple messages
- how to define your own message types

Topics

Lesson 4 Overview

The files for this lesson are located in the directories:

Windows:

```
%RTHOME%\java\tutorial\lesson4
```

UNIX:

```
$RTHOME/java/tutorial/lesson4
```

In the previous lesson, the receiving programs read messages using TipcSrv.next to read the next message in a program's incoming message queue and then operated on the message. To make full use of SmartSockets and its object-oriented features, it is important that you learn about SmartSockets callbacks. Before showing you some sample programs, the initial part of this lesson introduces the concept of callbacks.

Introduction to Callbacks

SmartSockets makes heavy use of callbacks to allow you visibility into its internal processing. A callback is a mechanism that allows you to be notified when a specified event occurs, such as a message is placed into the input queue. The callback can be associated with a certain event or with all events of a certain type. Callbacks that are associated with all events of a given type are called global callbacks. When the event occurs, the specified callback object's method is invoked to notify you that the event happened.

Your SmartSockets Java RTclient application contains classes that implement callback interfaces (either directly or through inheritance) and the specific code required for each handled event.

Table 2 summarizes the callback interfaces.

Table 2 Callback Interfaces

Callback Interface	Function
TipcCreateCb	The `create` method is called when an RTserver connection is successfully created
TipcDefaultCb	The `handle` method is called when no Process callback for a received message type is registered (see TipcProcessCb).
TipcDestroyCb	The `destroy` method is called when an RTserver connection is destroyed.
TipcErrorCb	The `error` method is called when certain SmartSockets errors occur.
TipcProcessCb	The `process` method is called when a message of a previously-specified matching type or subject is received.
TipcReadCb	The `read` method is called when a message is read and placed in the input queue.
TipcWriteCb	The `write` method is called when a message is written and removed from the output queue.

Only a very brief introduction to callback classes is provided in these lessons. See the online Javadoc format documentation for more detailed information about the specifics of SmartSockets Java callbacks.

All SmartSockets callback interfaces have some common characteristics:

- They are used to affect the processing of a message or simply to receive notification of some event.
- They have an associated priority. Priorities determine the order in which callback methods are invoked. Higher priority callbacks are called before lower priority ones. If two callbacks have the same priority, their relative calling order is undetermined. The getPriority and setPriority methods of the TipcCb class are used to retrieve and assign, respectively, callback priorities.
- They are uniquely identified by method and argument within a given application. This means that two callbacks cannot be registered to use the same callback class and argument within the same program. Attempts to add the same callback multiple times are ignored.

- Their action methods allow for an extra argument (of type Object) to be supplied when invoked. This parameter is not used by SmartSockets and is a convenient place to pass application-specific data to the callback operation.

Creating Callbacks

A callback is created by implementing the appropriate interface and then registering an instance of the event-handling class with the add*Type*Cb method of the active TipcSrv or TipcConn object. For example, to create an error callback (for example, `MyErrorCb`) that gets invoked when a non-recoverable error occurs on your program's connection to RTserver, a class needs to be created that implements the TipcErrorCb interface, as illustrated below:

```
public class MyErrorCbClass implements TipcErrorCb {
public error(int errNum, String errStr, Object obj) {
 // error handling here
   . . .
}
}
```

Keep in mind that this does not necessarily have to be a new, single-purpose object; Java classes can easily implement multiple interfaces.

You also need to add code similar to this to the main program to register the new callback with SmartSockets:

```
// the add method returns a reference to the callback
// used later; you don't want this to go out of scope,
// so this next line, the reference declaration, would
// probably be placed at the class level, if this class
// isn't transient
TipcCb MyErrorCbRef;

// in this example, srv is the current TipcSrv object
MyErrorCbClass MyErrorCb = new MyErrorCbClass;
MyErrorCbRef = srv.addErrorCb(MyErrorCb, srv);
if (null == MyErrorCbRef) {
 // error
}
```

In general, you should always check to make sure the callback is successfully registered. You also need to write the code for the error method in `MyErrorCbClass` to actually handle the event.

Manipulating Callbacks

To manipulate the various attributes of a callback, a handle (of type TipcCb) to the callback must first be acquired. An application retrieves this handle in one of two ways. First, the handle can be produced with the lookup*Type*Cb method of a TipcConn or TipcSrv object, where *Type* is replaced with one of: Create, Destroy, Default, Process, Error, Read, or Write. Alternately, the return value of the various callback add methods can be used; it is also the TipcCb corresponding to the added callback. Event callback properties can then be manipulated using the SmartSockets TipcCb.* utility functions:

getArgument	returns the callback's `Object` argument.
getCallback	returns the class that implements this callback's interface.
getPriority	returns this callback's priority.
setArgument	sets the callback's `Object` argument to the specified value.
setCallback	sets the class that implements this callback's interface.
setPriority	sets the callback's priority.

See the online documentation of the TipcCb class for more details on these methods.

Destroying Callbacks

Event callbacks may be unregistered by calling the remove*Type*Cb method with the TipcCb reference returned by the add*Type*Cb method.

Callback Types

There are a number of different callback types. In this lesson, you work with callbacks that are associated with a program's connection to RTserver. The next paragraphs describe the different types of callback interfaces available in SmartSockets. Following the descriptions, there are several example programs illustrating the use of callbacks.

This section presents a description of the callbacks and the methods used to register them. When possible, an example using the method is also presented.

Process Callbacks

Message process callbacks are invoked by SmartSockets when explicitly processing a message with the `process` method (defined in TipcConnClient and inherited by TipcSrv) or within the context of a `mainLoop` method, which also calls `process` internally. A `process` callback can be notified for a specific type of message or for all message types (see Subject Callbacks on page 75).

For example, a `process` callback can be created to respond only to the NUMERIC_DATA message type. When any message of that type is processed by calling TipcSrv.process(), the process callback's process method is fired. If a global process callback is created, it is fired for all NUMERIC_DATA type messages as well as for any other type of message.

In this example:

```
// srv is the active TipcSrv object
callbackRef = srv.addProcessCb(my_class, mt, srv);
if (null == callbackRef) {
  // error
}
```

the `my_class` argument is an instance of a class in your application that implements the TipcProcessCallback interface. The argument `mt` is a TipcMt object and is typically set by making a call to TipcSvc.mtLookup. If the second argument is `null`, a global process callback, called for all message types, is created. The final argument is of type `Object` and can be anything useful to your application. The TipcSrv object `srv` is specified.

Subject Callbacks

Message subject callbacks are invoked by SmartSockets when explicitly processing a message with the `process` method (defined in TipcConnClient and inherited by TipcSrv) or within the context of a `mainLoop` method, which also calls `process` internally. This type of callback is the most frequently used. Subject callbacks operate in a manner very similar to process callbacks except that the function executed is selected based on the message's destination, not its type. A subject callback can be executed when a message is received for a specific destination (remember that a subject is used as the value of a message's destination property) or for all message destinations. Just as with process callbacks, you can define a default callback to be executed if no callback has been defined for a given subject.

For example, a subject callback can be created to respond only to the `/stocks/computer` subject. When any message with that destination is processed by calling TipcSrv.process(), the subject callback's `process` method is fired. If a global subject callback is created, it is fired for all messages with the subject `/stocks/computer` as well as for messages with any other subject.

In this example:

```
// srv is the active TipcSrv object
callbackRef = srv.addProcessCb(my_class,"/stocks/computer",srv);
if (null == callbackRef) {
 // error
}
```

the `my_class` argument is an instance of a class in your application that implements the TipcSubjectCallback interface. The second argument is a string specifying the subject. If the second argument is `null`, a global subject callback (called for all subjects) is created. The final argument is of type `Object` and can be anything useful to your application. The TipcSrv object `srv` is specified.

You can specify type as well as subject for a callback. This means that there are four possible scenarios for callback execution. A callback can be defined to execute for:

- All message types addressed to one subject
- One message type addressed to one subject
- One message type addressed to any subject
- All message types addressed to any subject

Default Callbacks

Message default process callbacks are invoked by SmartSockets when processing a message with the `process` method (within `mainLoop`) if a specific process callback has not been registered. Default process callbacks are useful for processing unexpected message types or for generic processing of most message types. For example:

```
// srv is the active TipcSrv object
callbackRef = srv.addDefaultCb(my_class,srv);
if (null == callbackRef) {
 // error
}
```

Read Callbacks

The `read` method, addReadCb(), is executed when an incoming message is read from RTserver into the read buffer of the program and first unpacked into a message. Read callbacks are most commonly used for writing incoming messages to message files.

Write Callbacks

The `write` method, addWriteCb(), is executed when an outgoing message is sent to RTserver. Write callbacks are most commonly used for writing outgoing messages to message files.

Server Create Callbacks

The server `create` method, addCreateCb(), is called when RTclient connects or reconnects to RTserver. It can be useful for performing security checks such as process authentication.

Server Destroy Callbacks

The `destroy` method, addDestroyCb(), is called when RTclient destroys its connection to RTserver. Server destroy callbacks are useful for RTclients that need to know when the connection to RTserver has been broken for any reason.

Error Callbacks

The `error` method, addErrorCb(), is executed when an unrecoverable error occurs. These errors include problems with the connection to RTserver and network failures such as:

- a write timeout has occurred
- a read operation has failed

  The most common occurrence of this error is when RTserver destroys its connection with the program (that closes the connection).
- a write operation has failed

  The most common occurrence of this error is when the RTserver destroys its connection with the program (that closes the connection).

Using Callbacks

In this section you modify your examples from previous lessons to use process and default callbacks.

Writing a Process Callback

To see a callback in action, define a message process callback object to operate on incoming NUMERIC_DATA messages. Process callback objects are the most common way in SmartSockets to perform the main processing of a message.

The next section describes a callback implementation in detail. This callback object, whose `process()` method is invoked when a message of type NUMERIC_DATA is processed with TipcSrv.process() or using TipcSrv.mainLoop(), simply accesses and prints the fields of the message. There is another example of a process callback in Processing of GMD_FAILURE Messages on page 182.

The files for this lesson are located in the directories:

Windows:

```
%RTHOME%\java\tutorial\lesson4
```

UNIX:

```
$RTHOME/java/tutorial/lesson4
```

Step 1 **Copy the receive.java program**

Copy the `receive.java` program into your working directory. This is an example of the `receive.java` program:

```
//-------------------------------------------------------------------
// receive.java -- output a NUMERIC_DATA with callback

import java.io.*;
import com.smartsockets.*;

public class receive {

  public class receiveCb implements TipcProcessCb {

    public void process(TipcMsg msg, Object arg) {
      System.out.println("Received NUMERIC_DATA message.");

 // position the field ptr to the beginning of the message
     try {
       msg.setCurrent(0);
  }    catch (TipcException e) {
         Tut.fatal(e);
        } // catch

        System.out.println("Contents of NUMERIC_DATA message:");

  // read the data part of the message
        try {
          String var_name;
          while (true) {
            var_name = msg.nextStr();
            double var_value;
            var_value = msg.nextReal8();
            System.out.println("Var name = " + var_name +
            ", value = " + var_value);
          } // while
        } catch (TipcException e) {
   // catch end-of-message-data exception, do nothing.
          } // catch
        } // process
      } // receiveCb

      public receive() {
        TipcMsg msg = null;

  // set the project
        try {
          Tut.setOption("ss.project", "smartsockets");
          TipcSrv srv=TipcSvc.getSrv();

  // connect to RTserver
        if (!srv.create()) {
          Tut.exitFailure("Couldn't connect to RTserver!");
        } // if
```

```
       // subscribe to the appropriate subject
23        srv.setSubjectSubscribe("/ss/tutorial/lesson4", true);

       // create a new receive callback and register it
24        receiveCb rcb = new receiveCb();
25        TipcCb rcbh = srv.addProcessCb(rcb, TipcMt.NUMERIC_DATA,
                                          srv);
       // check the 'handle' returned for validity
26        if (null == rcbh) {
27          Tut.exitFailure
                      ("Couldn't register process listener!");
          } // if

       // read and process a message
28        msg = srv.next(TipcDefs.TIMEOUT_FOREVER);

       // all callbacks are triggered by TipcSrv's process()
       // method
29        srv.process(msg);

       // clean up and disconnect from RTserver
30        srv.removeProcessCb(rcbh);
31        srv.destroy();
32      } catch (TipcException e) {
33        Tut.fatal(e);
        } // catch
      } // receive (constructor)

34    public static void main(String[] argv) {
35      receive r = new receive();
      } // main
    } // receive
```

For this example, the bulk of the code has been moved to the constructor for the `receive` class, and `main` simply instantiates a `receive` object to begin operation. While examining the `receive` constructor, the first thing to notice is that the processing of the NUMERIC_DATA message has been moved out of this section of code and into the callback class, `receiveCb`, lines 4-17. A call to the method TipcSrv.process() is also added on line 29 to invoke the callback when it is time to process the message.

Step 2 Copy the send.java program and compile

Copy the `send.java` program into your working directory, and then compile the receiving and sending programs:

```
$ javac receive.java
$ javac send.java
```

Step 3 **Ensure the RTserver is running**

Make sure RTserver is running. If not, start it:

```
$ rtserver
```

On platforms that support both 32- and 64-bit, use the `rtserver64` command to run the 64-bit version of the `rtserver` script.

Step 4 **Start the receiving program**

Run the receiving program using:

```
$ java receive
```

Step 5 **Start the sending program**

After a few moments, run the sending program in a second window:

```
$ java send
```

This output is displayed by the receiving program:

```
Received NUMERIC_DATA message.

Contents of NUMERIC_DATA message:
---------------------------------
Var name = X, Value = 10.0
Var name = Y, Value = 20.0
Var name = Z, Value = 30.0
```

Writing a Default Callback

In the previous section, the example was set up to invoke a callback when a NUMERIC_DATA message is processed. What happens if you send a message that is not of type NUMERIC_DATA? Next you try it and find out.

Step 6 **Copy the send2.java program**

Copy the `send2.java` program into your working directory.

This program is the equivalent of modifying the original sending program by adding these lines after connecting to RTserver and before creating the NUMERIC_DATA message:

```
TipcMsg msgi = TipcSvc.createMsg(TipcMt.INFO);
msgi.setDest("/ss/tutorial/lesson4");
msgi.appendStr("Hello World!");
srv.send(msgi);
srv.flush();
```

This new code sends an INFO message to your receiving program, followed by a NUMERIC_DATA message.

Note that in the next three steps, you run `receive` with `send2` instead of the usual pairing of `receive` with `send` or `receive2` with `send2`.

Step 7 **Compile the send2.java program**

Compile the `send2.java` program using the command:

```
$ javac send2.java
```

Step 8 **Start the receiving program**

Start the original receiving program in one window of your display using the command:

```
$ java receive
```

Step 9 **Start the new sending program**

After a few moments, run the new sending program (which sends an INFO message) in another window using the command:

```
$ java send2
```

You do not see any output in the window where you ran the receiving program because the INFO message was received before the NUMERIC_DATA message. Because there was no callback created to process a message of type INFO, the message was ignored. The second message was sent, but because the receiving program is set up to read and process only one message, the NUMERIC_DATA message was never read.

Step 10 **Copy the receive2.java program**

Copy the `receive2.java` program into your working directory.

The `receive2.java` program is simply `receive.java`, modified so that it can read and process any number of messages. Copying this file is the equivalent of replacing lines 28 and 29 of the receiving program with this piece of code:

```
// Read and process all incoming messages
while (null != (msg = srv.next(TipcDefs.TIMEOUT_FOREVER))) {
srv.process(msg);
} // while
```

This code creates a `while` loop that continues to read and process messages until TipcSrv.next returns `null`.

Now you should create a default callback to process any non-NUMERIC_DATA messages by adding this code to the receiving program after the callback for NUMERIC_DATA messages has been registered:

```
// register receiveCallback again as a default callback
TipcCb dcbh = srv.addDefaultCb(rcb, srv);
```

To complete the program, the default callback method `handle` should be added to the `receiveCb` class. This method simply prints out the name and type of the message. These changes have been made in the `receive2.java` program:

```
//----------------------------------------------------------------
// receive2.java -- output a NUMERIC_DATA message with a callback

import java.io.*;
import com.smartsockets.*;

public class receive2 {

  public class receiveCb
  implements TipcProcessCb, TipcDefaultCb {

    public void process(TipcMsg msg, Object arg) {
      System.out.println("Received NUMERIC_DATA message");

   // position the field ptr to the beginning of the message
      try {
        msg.setCurrent(0);
      } catch (TipcException e) {
          Tut.fatal(e);
      } // catch

      //  read the data part of the message
      try {
        String var_name;
        while (true) {
          var_name = msg.nextStr();
          double var_value;
          var_value = msg.nextReal8();
          System.out.println("Var name = " + var_name +
                                 ", value = " + var_value);
        } // while
        } catch (TipcException e) { }
    // catch end-of-message-data exception, do nothing.
      } // process

   // handle() is for responding to default messages
      public void handle(TipcMsg msg, Object arg) {
        System.out.println("Receive: unexpected message type name" +
        " is <" + msg.getType().getName() + ">");
      } // handle
    } // receiveCb

    public receive2() {
      TipcMsg msg = null;
   // set the ss.project
      try {
        Tut.setOption("ss.project", "smartsockets");
        TipcSrv srv=TipcSvc.getSrv();
```

```
     // create a new receive listener and register it
           receiveCb rcb = new receiveCb();
           TipcCb rcbh = srv.addProcessCb(
           rcb, TipcSvc.lookupMt(TipcMt.NUMERIC_DATA), srv);
     // check the 'handle' returned for validity
           if (null == rcbh) {
             Tut.exitFailure("Couldn't register process listener!");
           } // if

     // register receiveCb again as a default listener
           TipcCb dcbh = srv.addDefaultCb(rcb, srv);
     // check the 'handle' returned for validity
           if (null == dcbh) {
             Tut.exitFailure("Couldn't register default listener!");
           } // if

     // connect to RTserver
           if (!srv.create()) {
             Tut.exitFailure("Couldn't connect to RTserver!");
           } //  if

     // subscribe to the appropriate subject
           srv.setSubjectSubscribe("/ss/tutorial/lesson4", true);

     // read and process all incoming messages
           while (null != (msg = srv.next(TipcDefs.TIMEOUT_FOREVER))) {
               srv.process(msg);
           } // while

     // disconnect from RTserver
           srv.destroy();

     // unregister the listeners for completeness
           srv.removeProcessCb(rcbh);
           srv.removeDefaultCb(dcbh);
         } catch (TipcException e) {
           Tut.fatal(e);
       } // catch
     } // receive2 (constructor)

     public static void main(String[] argv) {
        receive2 r = new receive2();
     } // main
   } // receive2 class
```

Before running your updated receiving program, copy the `send3.java` program to your working directory. The `send3.java` program is the `send2.java` program, modified to send multiple messages.

This is the `send3.java` program:

```
//-------------------------------------------------------------------
// send3.java -- write an INFO and then NUMERIC_DATA messages

import java.io.*;
import com.smartsockets.*;

  public class send3 {

    public static void main(String[] argv) {
      try {
        Tut.setOption("ss.project", "smartsockets");

        TipcSrv srv=TipcSvc.getSrv();
        if (!srv.create()) {
          Tut.exitFailure("Couldn't connect to RTserver!");
        } // if

    // send a message of type INFO
        TipcMsg msgi = TipcSvc.createMsg(TipcMt.INFO);
        msgi.setDest("/ss/tutorial/lesson4");
        msgi.appendStr("Hello World!");
        srv.send(msgi);
        srv.flush();

    // create a message of type NUMERIC_DATA
         TipcMsg msg = TipcSvc.createMsg(TipcMt.NUMERIC_DATA);

    // set the destination subject of the message
         msg.setDest("/ss/tutorial/lesson4");

    // each time through the loop send a NUMERIC_DATA
    // message with three values
         for (int i = 0; i < 30; i = i + 3) {
           msg.setNumFields(0);
           msg.appendStr("X");
           msg.appendReal8(i);
           msg.appendStr("Y");
           msg.appendReal8(i+1.0);
           msg.appendStr("Z");
           msg.appendReal8(i+2.0);

     // send the message
           srv.send(msg);
           srv.flush();
         }

     // disconnect from RTserver
           srv.destroy();
         } catch (TipcException e) {
           Tut.warning(e);
         } // catch
      } // main
    } // send3
```

Let's examine the key lines in this program:

Lines 16-25 This is a `for` loop that sends out a series of NUMERIC_DATA messages.

Line 17 The same message is re-used each time; only the data part of the message is changed. At the beginning of the loop, TipcMsg.setNumFields resets the data part of the message to have zero fields.

In the next few steps, you run `receive2` with `send3` instead of the usual pairing of `receive2` with `send2` or `receive3` with `send3`.

Step 11 **Copy the send3.java program**

Copy the `send3.java` program into your working directory, and compile it with the command:

```
$ javac send3.java
```

Step 12 **Compile the new receive2.java program**

Compile your new `receive2.java` program using the command:

```
$ javac receive2.java
```

Step 13 **Start the receiving program**

Start the receiving program in one window of your display using the command:

```
$ java receive2
```

Step 14 **Start the new sending program**

In another window, to send a message to the receiving program, run the new sending program using the command:

```
$ java send3
```

After running the sending program, this output is displayed in the receiving program window:

```
Receive: unexpected message type name is <info>
Received NUMERIC_DATA message
Var name = X, value = 0.0
Var name = Y, value = 1.0
Var name = Z, value = 2.0
Received NUMERIC_DATA message
Var name = X, value = 3.0
Var name = Y, value = 4.0
Var name = Z, value = 5.0
```

```
// ... Output omitted here ...
Received NUMERIC_DATA message
Var name = X, value = 27.0
Var name = Y, value = 28.0
Var name = Z, value = 29.0
```

When the `send3` program has completed, notice that the `receive2` program is still hanging; it is waiting for more messages.

Step 15 **Interrupt the receiving program**

Type Ctrl-c to interrupt the `receive2` program.

For each NUMERIC_DATA message received, the callback method `receiveCb.process()` was invoked to print out the contents of the data part of the message. The very first message received was an INFO message. Because there were no process callbacks available for INFO messages, the default callback, `receiveCb's handle` method, was called and printed the type of unexpected message received.

Writing a Subject Callback

Rather than processing a message based on its type, you can process a message based on its destination using subject callbacks. 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.

To create a subject callback, you invoke one of TipcSrv's `addProcessCb` method's overloaded forms that allow a `String` subject to be specified, as shown:

`addProcessCb(`*callback, mt, subject, arg*`)`
`addProcessCb(`*callback, subject, arg*`)`

where

subject is the destination you wish to specify the callback on and

mt is the message type the callback should be applied to.

You can specify a value of `null` for *subject* or *mt* to specify "any." (It may be necessary to explicitly cast `null` as a String so the compiler can determine which method implementation to use.) Subject callbacks are actually a superset of process callbacks as they allow message type and subject callbacks to be mixed (for example, execute this callback when a message of type T arrives on subject S).

Some examples of creating subject callbacks are shown:

```
TipcSrv srv = TipcSvc.getSrv();
TipcMt mt = TipcSvc.lookupMt(TipcMt.INFO);
```

```
// Call subj_cb's process() method upon receipt of any
// message that has a destination of "/tutorial"
srv.addProcessCb(subj_cb, "/tutorial", null);

// Execute subj_cb's process() method upon receipt of any
// messages of type INFO, regardless of the destination
srv.addProcessCb(subj_cb, mt, (String)null, null);

// Execute the function subj_cb for any messages of type
// INFO, which have a destination of "/tutorial"
srv.addProcessCb(subj_cb, mt, "/tutorial", null);
```

In this section you modify the examples used for process callbacks to show how easy it is to use subject callbacks. The next code example describes a specific subject callback in detail. The callback object's `process` method is invoked when a message is received that has a destination of `/ss/tutorial/lesson4`. The `process` method simply gets the type of the message and then prints the fields of the message.

Step 16 Copy the subjcbs.java program

Copy the subject callback program, `subjcbs.java`, into your working directory. The contents of the file `subjcbs.java` are:

```
//-----------------------------------------------------------------
// subjcbs.java -- output messages through subject callbacks

import java.io.*;
import com.smartsockets.*;

public class subjcbs {

  public class processLesson4 implements TipcProcessCb {

    public void process(TipcMsg msg, Object arg) {
      System.out.println("*** Received message of type <" +
                         msg.getType().getName()+">");

   // position the field ptr to the beginning of the message
      try {
        msg.setCurrent(0);
      }
      catch (TipcException e) {
        Tut.fatal(e);
       }
```

```
      // display message contents based on type
        int mt = msg.getType().getNum();
        switch (mt) {
          case TipcMt.INFO:
            try {
              System.out.println("Text from message = "+
                                      msg.nextStr());
            } catch (TipcException e) { }
            break;

            case TipcMt.NUMERIC_DATA:
              String var_name;
              try {
         // display the repeating part of NUMERIC_DATA message
                while (true) {
                  var_name = msg.nextStr();
                  double var_value;
                  var_value = msg.nextReal8();
                  System.out.println("Var name = " + var_name +
                              ", value = " + var_value);
              } // while
        // catch end-of-message-data exception, do nothing.
            } catch (TipcException e) { }
            break;

          default:
       // handle messages of unknown type
          System.out.println("Unable to process messages of this type!");
           break;
        } // switch
      } // process
    } // processLesson4

    public subjcbs() {
      TipcMsg msg = null;

    // set the ss.project
      try {
        Tut.setOption("ss.project", "smartsockets");
        TipcSrv srv=TipcSvc.getSrv();

     // create a new receive SUBJECT callback and register it
        processLesson4 pl = new processLesson4();
        TipcCb rcbh = srv.addProcessCb(pl, "/ss/tutorial/lesson4", srv);
     // check the 'handle' returned for validity
        if (null == rcbh) {
          Tut.exitFailure("Couldn't register subject callback!");
        } // if

     // connect to RTserver
        if (!srv.create()) {
          Tut.exitFailure("Couldn't connect to RTserver!");
        } // if
     // subscribe to the appropriate subject
        srv.setSubjectSubscribe("/ss/tutorial/lesson4", true);
```

```
     // read and process all incoming messages
40       while (srv.mainLoop(TipcDefs.TIMEOUT_FOREVER)) {
         } // while

     // unregister the callbacks
41       srv.removeProcessCb(rcbh);

     // disconnect from RTserver
42       srv.destroy();
43     } catch (TipcException e) {
44       Tut.fatal(e);
       } // catch
     } // subjcbs (constructor)

45   public static void main(String[] argv) {
46     new subjcbs();
     } // main
   } // subjcbs class
```

Some interesting things to learn from your new `subjcbs` program are:

Lines 5-28 The processing of messages of all types is now in the callback object `ProcessLesson4` process method. The method first gets the type of the message and then prints outs the contents based on the type. In effect, you have a simple process (message type) callback within a subject callback.

Lines 11-12 The received message's type is extracted and acted upon with a switch statement.

Lines 33-34 A subject callback object, `pl`, is created and registered for messages arriving with a destination of `/ss/tutorial/lesson4`.

Line 39 Even though we have defined a subject callback on `/ss/tutorial/lesson4`, we still need to make sure that the program subscribes to the subject.

Line 40 TipcSrvMainLoop invokes the subject callback whenever a message arrives with the given destination.

You now execute the new program using subject callbacks to verify that it works correctly.

Step 17 **Copy the subjcbs.java program and compile**

Copy the `subjcbs.java` program into your working directory, and compile it with the command:

```
$ javac subjcbs.java
```

Step 18 Start the subject callback program

Start the subject callback program in one window of your display using the command:

```
$ java subjcbs
```

Step 19 Start the sending program

In another window, run the sending program used earlier in this lesson with the command to send a message to the subject callback program:

```
$ java send3
```

After running the sending program, this output is displayed in the window where you ran the subject callback program:

```
Attempting connection to <tcp:_node:5101> RTserver.
Connected to <tcp:_node:5101> RTserver.
*** Received message of type <info>
Text from message = Hello World!
*** Received message of type <numeric_data>
Var name = X, value = 0.0
Var name = Y, value = 1.0
Var name = Z, value = 2.0
*** Received message of type <numeric_data>
Var name = X, value = 3.0
Var name = Y, value = 4.0
Var name = Z, value = 5.0
```

// more output omitted here...

```
*** Received message of type <numeric_data>
Var name = X, value = 27.0
Var name = Y, value = 28.0
Var name = Z, value = 29.0
```

When the sending program has completed, notice that the subject callback program is still hanging. It is waiting for more messages.

Step 20 Interrupt the subject callback program

Type Ctrl-c to interrupt the subject callback program.

For each message received, the callback object `ProcessLesson4`'s `process` method was invoked to print out the contents of the data part of the message, regardless of the type of the message.

Specifying a Callback Based on Subject and Message Type

The example in the previous section can be further modified to specify a different subject callback for each of the different message types: INFO and NUMERIC_DATA. This is done by creating two new callback objects: `ProcessInfo` and `ProcessNumData`. In the main program, two calls are required to TipcSrv.addProcessCb, one for each of the message types. The complete example is shown:

```
//----------------------------------------------------------------
// subjcbs2.java -- output messages through subject/mt callbacks

import java.io.*;
import com.smartsockets.*;

public class subjcbs2 {

  public class processInfo implements TipcProcessCb {

      public void process(TipcMsg msg, Object arg) {
        System.out.println("*** Received INFO message");
        try {
          msg.setCurrent(0);
          System.out.println("Text from message = " + msg.nextStr());
        } catch (TipcException e) { }
      } // process
    } // processInfo

    public class processNumData implements TipcProcessCb {

      public void process(TipcMsg msg, Object arg) {
        System.out.println("*** Received NUMERIC_DATA message");
        String var_name;
        try {
          msg.setCurrent(0);
     // display the repeating part of NUMERIC_DATA message
          while (true) {
             var_name = msg.nextStr();
             double var_value;
             var_value = msg.nextReal8();
             System.out.println("Var name = " + var_name +
                                 ", value = " + var_value);
          } // while
      // catch end-of-message-data exception, do nothing.
      } catch (TipcException e) { }
      } // process
    } // processNumData

    public subjcbs2() {
      TipcMsg msg = null;

  // set the ss.project
      try {
        Tut.setOption("ss.project", "smartsockets");
        TipcSrv srv=TipcSvc.getSrv();
```

```
      // create a new info mt/subject callback and register it
           processInfo pi = new processInfo();
           TipcCb rcbh1 = srv.addProcessCb(pi,
                   TipcSvc.lookupMt(TipcMt.INFO),
               "/ss/tutorial/lesson4", null);
      // check the 'handle' returned for validity
           if (null == rcbh1) {
             Tut.exitFailure("Couldn't register subject callback!");
           } // if

      // create a new info mt/subject callback and register it
           processNumData pnd = new processNumData();
           TipcCb rcbh2 = srv.addProcessCb(pnd,
                            TipcSvc.lookupMt(TipcMt.NUMERIC_DATA),
                            "/ss/tutorial/lesson4", null);
      // check the 'handle' returned for validity
           if (null == rcbh2) {
             Tut.exitFailure("Couldn't register subject callback!");
           } // if

      // connect to RTserver
           if (!srv.create()) {
             Tut.exitFailure("Couldn't connect to RTserver!");
           } // if
      // subscribe to the appropriate subject
           srv.setSubjectSubscribe("/ss/tutorial/lesson4", true);

      // read and process all incoming messages
           while (srv.mainLoop(TipcDefs.TIMEOUT_FOREVER)) {
           } // while

      // unregister the callbacks
           srv.removeProcessCb(rcbh1);
           srv.removeProcessCb(rcbh2);

      // disconnect from RTserver
           srv.destroy();
         } catch (TipcException e) {
           Tut.fatal(e);
         } // catch
       } // subjcbs2 (constructor)

       public static void main(String[] argv) {
         new subjcbs2();
       } // main

    } // subjcbs2 class
```

For more details on subject and message type callbacks, see the *TIBCO SmartSockets User's Guide*.

Using the TipSrv.mainLoop Convenience Method

In the `receive2` program, this while loop is added to read and process all incoming messages:

```
// read and process all incoming messages
while (null != (msg = srv.next(TipcDefs.TIMEOUT_FOREVER))) {
       srv.process(msg);
     } // while
```

This entire loop can be replaced by this single call:

```
srv.mainLoop(TipcDefs.TIMEOUT_FOREVER);
```

The TipcSrv.mainLoop() convenience method receives and processes messages from RTserver by calling TipcSrv.next and TipcSrv.process over and over. TipcSrv.mainLoop is a convenience method that keeps calling TipcSrv.next with the time remaining from *timeout* until TipcSrv.next returns `false`. For each message that TipcSrv.mainLoop gets, it processes the message with TipcSrv.process. Use `0.0` for *timeout* to poll the RTserver connection and catch up on all pending messages that have accumulated or to return immediately if no messages are pending. Use `TipcDefs.TIMEOUT_FOREVER` for *timeout* to read and process messages indefinitely. See the online documentation on TipcSrv.mainLoop for more details.

A modified `receive2` program, which uses TipcSrv.mainLoop, is located in the file `receive3.java`. You can compile and run it with `send3` if you want to verify that it produces the same output as before.

Using Server Create and Destroy Callbacks

Earlier in this lesson, you saw example programs that used process and default callbacks to work with messages. In this section two new callback types are shown: server create and server destroy. A server create callback's `create` method is executed when an RTclient connects to RTserver, and a server destroy callback's `destroy` method is executed when an RTclient disconnects from RTserver.

In this lesson, you trigger these callbacks with a simple example. The program, `srvcbs`, prompts you for a password each time it tries to connect to RTserver. If the password is incorrect, the program is disconnected from RTserver and terminated.

Step 21 **Copy the svrcbs.java file**

Copy the create callback program, `svrcbs.java`, into your working directory.

This is the `svrcbs.java` program:

```
//-----------------------------------------------------------
// svrcbs.java -- server create/destroy callbacks

import java.io.*;
import com.smartsockets.*;

public class svrcbs {
  String password_correct = "ssjava";

  public class serverConnect implements TipcCreateCb {

     public void create(Object srv_obj) {
       TipcSrv srv = (TipcSrv)srv_obj;
       System.out.println("Connecting to RTserver...");
       System.out.print("Please enter password: ");
       BufferedReader in = new BufferedReader(
           new InputStreamReader(System.in) );
       String password_entered = null;
       try {
            password_entered = in.readLine();
       } catch (IOException e) {
         System.out.println("Error! "+e.getMessage());
       } // catch
       if (password_entered.equals(password_correct)) {
         System.out.println("Password accepted!");
       }
       else {
         System.out.println("Password is not correct! " +
           "You are being disconnected from RTserver");
         try {
           srv.destroy();
           Tut.exitSuccess();
         } catch (TipcException e) {
           Tut.warning("Can't destroy server connection: " +
                e.getMessage());
         } // catch
       } // else
     } // create
  } // serverConnect
```

```
22  public class serverDisconnect implements TipcDestroyCb {
23    public void destroy(Object obj) {
24      System.out.println("...Disconnecting from RTserver");
      } // destroy
    } // serverDisconnect

25  public svrcbs() {
26    TipcMsg msg = null;

      try {
27      TipcSrv srv = TipcSvc.getSrv();

    // create a new connect callback and register it
28      serverConnect sc = new serverConnect();
29      TipcCb sch = srv.addCreateCb(sc, srv);
    // check the 'handle' returned for validity
30      if (null == sch) {
31         Tut.exitFailure("Couldn't register create callback!");
        } // if
    // and a destroy callback
32      serverDisconnect sd = new serverDisconnect();
33      TipcCb sdh = srv.addDestroyCb(sd, srv);
    // check the 'handle' returned for validity
34      if (null == sdh) {
35        Tut.exitFailure("Couldn't register destroy callback!");
        } // if

    // connect to RTserver
36      srv.create();

    // read and process all incoming messages
37      while (true) {
38        srv.mainLoop(2.0);
        } // while

39    } catch (TipcException e) {
40      Tut.fatal(e);
      } // catch
      // svrcbs (constructor)

41   public static void main(String[] argv) {
42     new svrcbs();
    } // main
  } // svrcbs
```

Let's examine the key lines in this program:

Line 29 — The server create callback (`serverConnect`) is registered with the call to TipcSrv.addCreateCb.

Line 33 — The server destroy callback (`serverDisconnect`) is registered with the call to TipcSrv.addDestroyCb.

Line 36 — Notice that both the server create and server destroy callbacks are registered before the initial connection to RTserver is made through a call to TipcSrv.create.

Lines 17-21 The server create callback disconnects the program from RTserver with the TipcSrv.destroy method if an incorrect password is given.

Let's see how these programs use callbacks, and how RTserver affects their operation.

Step 22 **Compile the svrcbs.java program**

Compile the `svrcbs.java` program using the command:

```
$ javac svrcbs.java
```

Step 23 **Start the create callback program**

Start the create callback program in one window of your display using the command:

```
$ java svrcbs
```

This output is displayed:

```
Attempting connection to <_node>.
Connecting to RTserver...
Please enter password:
```

The last two lines of output are from the server create callback. This was executed when the process tried to connect to RTserver for the first time. You are prompted for a password.

Step 24 **Enter a password**

Enter this password and press the return key:

```
Please enter password: ssjava
```

When the correct password is entered, this text is displayed:

```
Password accepted!
```

The program is now successfully connected to RTserver. Let's manually break the connection to RTserver and see what happens.

Step 25 **Stop the RTserver**

In another window, stop RTserver using a command line argument to the `rtserver` command:

```
$ rtserver -stop
```

This new output is displayed in the window where you ran the create callback program:

```
WARNING: lost connection: reader: in: connection dropped Connection
reset
...Disconnecting from RTserver
Waiting before reconnecting.
Attempting connection to <_node> RTserver.
WARNING: lost connection: Connection refused: connect
Attempting connection to <_node> RTserver.
WARNING: lost connection: Connection refused: connect
Attempting connection to <_node> RTserver.
WARNING: lost connection: Connection refused: connect
Attempting connection to <_node> RTserver.
```

This output continues until another RTserver is found. Stopping RTserver resulted in a sequence of events happening:

1. The server destroy callback was executed and output:

   ```
   ...Disconnecting from RTserver
   ```

2. The create callback program then attempted to re-connect with RTserver. This is a fault-tolerant feature of SmartSockets.

Step 26 **Start a new RTserver**

In the other window (where you stopped RTserver), start a new RTserver:

```
$ rtserver
```

On platforms that support both 32- and 64-bit, use the `rtserver64` command to run the 64-bit version of the `rtserver` script.

When the connection is re-established, the server create callback is executed and you are again prompted for the password:

```
Connecting to RTserver...
Please enter password:
```

Step 27 **Enter an incorrect password**

This time, enter an incorrect password:

```
Please enter password: foo
```

This output is displayed:

```
Password is not correct!
You are being disconnected from RTserver
...Disconnecting from RTserver
```

In this case, the server create callback disconnected from RTserver and terminated the program. When disconnecting from RTserver, the server destroy callback was executed. This demonstrates how callbacks can trigger events to which other callbacks then respond.

Creating Your Own Message Types

SmartSockets comes with many predefined standard message types, such as NUMERIC_DATA. These standard message types are described in detail the *TIBCO SmartSockets User's Guide*.

When there is no standard message type to satisfy a requirement of your application, you can create your own (called a user-defined message type). Once you create it, 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 TipcSvc.createMt method. This example creates a message type named XYZ_COORD_DATA with fields (X, Y, and Z coordinates) that are 4-byte integers:

```
// type number must be greater than zero
static final int XYZ_COORD_DATA = 1001;
try {
    TipcMt xyzMt = TipcSvc.createMt("xyz_coord_data",
                                    XYZ_COORD_DATA, "int4 int4 int4");
    if (null == xyzMt) {
   // error
    } catch (TipcException e) {
      //  message type already exists or other error
    }
```

A message type is a template for a specific kind of message. Once the message type is created, any number of messages of that type can be created. The first argument to TipcSrv.createMt is the message type *name*, which should be an identifier (`String`). The second argument is the message type number, which is a signed four-byte integer (`int`). Message type numbers less than one are reserved for SmartSockets standard message types. The standard SmartSockets message types use similar names and numbers (for example, the message type with the name `numeric_data` has a defined number TipcMt.NUMERIC_DATA). The third argument to TipcMtCreate is the message type *grammar*, used to identify the layout of fields in messages that use this message type.

The message 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 of `real8`, which defines the first and only field as being an eight-byte real number. The table lists the primitive types that can appear as a field in the grammar of a standard or user-defined message type. Comments (delimited by */* */* or *(* *)*) are also allowed in the grammar.

Field Type	Meaning
`binary`	non restrictive array of characters (such as an entire data structure or the entire contents of a file)
`char`	1-byte integer

Field Type	Meaning
`int2`	2-byte integer
`int2_array`	array of `int2`
`int4`	4-byte integer
`int4_array`	array of `int4`
`int8`	8-byte integer
`int8_array`	array of `int8`
`msg`	a message
`msg_array`	array of `msg`
`real4`	4-byte real number
`real4_array`	array of `real4`
`real8`	8-byte real number
`real8_array`	array of `real8`
`str`	a C string (`'\0'`-terminated array of characters)
`str_array`	array of `str`
`xml`	xml object

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 ({}) can 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 per grammar.

Sample Programs

This section contains complete sample programs for creating, sending, reading, and processing a user-defined message type called XYZ_COORD_DATA.

Step 28 **Copy the snd_umsg.java and rcv_umsg.java programs**

Copy `snd_umsg.java` and `rcv_umsg.java` into your working directory.

This program creates and sends messages of user-defined type XYZ_COORD_DATA:

```
//-----------------------------------------------------------------
// snd_umsg.java - create and send a series of messages of
// user-defined type XYZ_COORD_DATA

import java.io.*;
import com.smartsockets.*;

public class snd_umsg {
  private static final int XYZ_COORD_DATA = 1001;

 public static void main(String[] argv) {
   try {
 // set the ss.project
     Tut.setOption("ss.project", "smartsockets");

 // get handle to the RTserver
     TipcSrv srv = TipcSvc.getSrv();

     if (!srv.create()) {
       Tut.exitFailure("Couldn't connect to RTserver!");
     }   // if

 // define new message type
     TipcMt mt = null;
     try {
       mt = TipcSvc.createMt("xyz_coord_data", XYZ_COORD_DATA,
                             "int4 int4 int4");
    } catch (TipcException e) {
       Tut.exitFailure("Message type already exists!");
    } // catch

 // create a nessage of type XYZ_COORD_DATA
    TipcMsg msg = TipcSvc.createMsg(mt);

 // set message destination
    msg.setDest("/ss/tutorial/lesson4");

    for (int i = 0; i < 30; i += 3) {
 // in order to re-use message, reset number of fields to 0
      msg.setNumFields(0);
      msg.appendInt4(i);
      msg.appendInt4(i + 1);
      msg.appendInt4(i + 2);
```

```
      // send and flush the message
22      srv.send(msg);
23      srv.flush();
      } // for

      // disconnect from RTserver
24    srv.destroy();
      } catch (TipcException e) {
25    Tut.fatal(e);
      } // catch
    } // main
} // snd_umsg
```

Let's examine some of the key lines in this program:

Line 4	Defines the unique number that identifies the new message type. This number must be used consistently in all programs that reference the user-defined message type.
Line 11	Creates the new message type. This call to TipcMtCreate must be included in all programs that refer to the new message type.
Line 15	Creates a message of type XYZ_COORD_DATA.
Lines 17-21	Build the data part of the new message.

This program publishes a series of ten XYZ_COORD_DATA messages.

Step 29 Compile the snd_umsg.java program

Compile the sending user-message program, `snd_umsg.java`, program using the command:

```
$ javac snd_umsg.java
```

You now need a program to read and process the messages of the new XYZ_COORD_DATA type. This program also needs to create the message type. It also needs to define a process callback for the new message type. This is an example of the receiving user-message program, `rcv_umsg.java` program:

```
//----------------------------------------------------------------
// rcv_umsg.java -- display contents of received user-defined
// messages (type XYZ_COORD_DATA)

import java.io.*;
import com.smartsockets.*;

public class rcv_umsg {
  static final int XYZ_COORD_DATA = 1001;

   public class processXYZ implements TipcProcessCb {

     public void process(TipcMsg msg, Object arg) {
       System.out.println("Received XYZ_COORD_DATA message");
```

```
      // position the field ptr to the beginning of the message
        try {
           msg.setCurrent(0);
        } catch (TipcException e) {
           Tut.fatal(e);
        } // catch

      // traverse message; print value from each field
        try {
          int field_value = msg.nextInt4();
          System.out.println("Field 1 Value = " + field_value);
          field_value = msg.nextInt4();
          System.out.println("Field 2 Value = " + field_value);
          field_value = msg.nextInt4();
          System.out.println("Field 3 Value = " + field_value);
        } catch (TipcException e) {
          Tut.warning("Expected 3 fields--bad XYZ_COORD_DATA!\n");
       } // catch
     } // process
   } // processXYZ

public class processDefault implements TipcDefaultCb {

 public void handle(TipcMsg msg, Object arg) {
   System.out.println("Receive: unexpected message type name" +
    " is <" + msg.getType().getName() + ">");
  } // handle
 } // processDefault

public rcv_umsg() {
 TipcMsg msg = null;

  try {
  // set the ss.project
   Tut.setOption("ss.project", "smartsockets");

  // get handle to the RTserver
   TipcSrv srv=TipcSvc.getSrv();

  // define new message type
   TipcMt mt = null;
   try {
     mt = TipcSvc.createMt("xyz_coord_data", XYZ_COORD_DATA,
                           "int4 int4 int4");
     } catch (TipcAlreadyExistsException e) {
       Tut.exitFailure("Message type already exists!");
   } // catch

   // create a new receive listener and register it
     processXYZ pcb = new processXYZ();
     TipcCb ph = srv.addProcessCb(pcb, mt, srv);
  // check the 'handle' returned for validity
   if (null == ph) {
     Tut.exitFailure("Couldn't register processXYZ listener!");
   } // if
```

```
      // create and register default listener
34      processDefault dcb = new processDefault();
35      TipcCb dh = srv.addDefaultCb(dcb, srv);

      // check the 'handle' returned for validity
36        if (null == dh) {
37          Tut.exitFailure("Couldn't register default listener!");
        } // if

      // connect to RTserver
38        if (!srv.create()) {
39          Tut.exitFailure("Couldn't connect to RTserver!");
        } // if

      // subscribe to the appropriate subject
40       srv.setSubjectSubscribe("/ss/tutorial/lesson4", true);

      // read and process all incoming messages
41       while (srv.mainLoop(TipcDefs.TIMEOUT_FOREVER)) {
       } // while

      // unregister the listeners for completeness
42       srv.removeProcessCb(ph);
43       srv.removeDefaultCb(dh);

      // disconnect from RTserver
44       srv.destroy();
45     } catch (TipcException e) {
46       Tut.fatal(e);
      } // catch
    } // rcv_umsg (constructor)

47 public static void main(String[] argv) {
48  new rcv_umsg();
   } // main
  } // rcv_umsg class
```

Let's examine the key lines of this program:

Line 4	Defines the unique number that identifies the new message type. This number must be used consistently in all programs that use the user-defined message type.
Line 27	Creates the new message type. This call to TipcMtCreate must be included in all programs that refer to the new message type.
Line 31	Registers the process callback to be used for messages of type XYZ_COORD_DATA.
Lines 6-18	This method is invoked when a message of type XYZ_COORD_DATA needs to be processed. It prints out the three numbers in the data part of the message.

Step 30 Compile the rcv_umsg.java program

Compile the `rcv_umsg.java` program using the command:

```
$ javac rcv_umsg.java
```

Step 31 Start the receiving user-message program

Start the receiving user-message program in one window of your display using the command:

```
$ java rcv_umsg
```

Step 32 Use the new sending user-message program to send messages

In another window, send a series of `XYZ_COORD_DATA` messages to the receiving user-message program, using the sending user-message program with the command:

```
$ java snd_umsg
```

When you start the sending user-message program, this output is displayed in the window where the receiving user-message program is being run:

```
Received XYZ_COORD_DATA message
Field 1 Value = 0
Field 2 Value = 1
Field 3 Value = 2
Received XYZ_COORD_DATA message
Field 1 Value = 3
Field 2 Value = 4
Field 3 Value = 5
Received XYZ_COORD_DATA message
Field 1 Value = 6
Field 2 Value = 7
Field 3 Value = 8
.
.
.
Received XYZ_COORD_DATA message
Field 1 Value = 27
Field 2 Value = 28
Field 3 Value = 29
```

When the sending user-message program has completed, notice that the receiving user-message program is still hanging. It is waiting for more messages.

Step 33 Interrupt the receiving user-message program

Type Ctrl-c to interrupt the receiving user-message program.

Summary

The key concepts covered in this lesson are:

- Callbacks are interfaces specifying methods to be executed when certain events occur.
- Callbacks give you an object-oriented approach to the advanced features of SmartSockets.
- SmartSockets provides a number of different types of callbacks including: subject, process, default, read, write, server create, server destroy, and error.
- The two most common types of callbacks used in a SmartSockets application are:
  - — process callbacks, specifying a `process` method that is invoked when a message of a given type is to be processed. In the Java library, this includes the functionality of subject callbacks in the C library.
  - — subject callbacks, specifying a `process` method that is invoked when a message addressed to a given subject is to be processed.
  - — default callbacks, specifying a `handle` method which is invoked when there is no subject callback available for the given message type.
- Subject, process, and default callbacks are invoked by a call to TipcSrv.process().
- Server create callbacks are executed whenever an RTclient connects or reconnects to RTserver. Server destroy callbacks are executed when an RTclient loses its connection to RTserver.
- The convenience method TipcSrv.mainLoop can be used to read and process incoming messages from RTserver. It replaces repeated calls to TipcSrv.next and TipcSrv.process.
- TipcMsg.setNumFields can be used to reset the data part of a message. This is useful if you want to re-use the message.
- In situations where there are no SmartSockets standard message types to meet your requirements, you can create user-defined message types.
- User-defined message types are created with a call to TipcSrv.createMt. This takes three arguments
  - — a name, which is an identifier
  - — a unique number (greater than zero)
  - — a grammar, which defines what the data part of the message looks like

- A user-defined message type must be defined consistently, with TipcSrv.createMt, in all programs that use the message type. Once defined, it is treated no differently from a SmartSockets standard message type.

Chapter 7 Lesson 5: TIBCO SmartSockets Options

In this lesson you learn about:

- how Java SmartSockets options are stored in a properties database
- how to set options, directly and from an options database
- how to load options from local storage and remote locations using a URL

Topics

Lesson 5 Overview

The files for this lesson are located in the directories:

Windows

```
%RTHOME%\java\tutorial\lesson5
```

UNIX

```
$RTHOME/java/tutorial/lesson5
```

This lesson describes RTclient option databases, as well as techniques for loading and manipulating options programmatically. SmartSockets uses these options extensively for configuring a Java RTclient's behavior. While many of the options an RTclient may care to set can easily be "hard-coded," the use of option databases allows a more flexible, dynamic method of program setup and control.

The options available to Java RTclients are described in Setting RTclient Options on page 137. Remember that these options and their values are case sensitive.

Option (Property) Databases

SmartSockets options are retrieved from Java Properties databases. These databases take the form of text files containing key-value pairs, one entry per line. The "key" string may contain periods (.) to indicate property hierarchies, and all SmartSockets options reside in the "`ss.`" hierarchy. For example, two of the standard SmartSockets option properties are defined in the property database like this:

```
ss.server_auto_connect: false
ss.project: foo
```

Note that option names and values are case sensitive and can be presented in any order. Option properties should appear only once in the property database; if multiple instances of an option appear with conflicting properties, the last key-value pair in the database is used, overriding any earlier settings. Option databases can contain values for the standard RTclient options, as well as for user-defined settings.

Utility Methods for Handling Options

The basic option-handling methods are contained in the Tut utility class. Individual options are also often represented by instances of the TipcOption class.

When using multiple RTserver connections created by the TipcSvc.createSrv factory method, use the TipcSrv option-handling methods to change the option setting for a TipcSrv object. If an option is not set for a TipcSrv object, the option's default value is used or the value set by the Tut option-handling methods is used.

This table summarizes the relevant Tut and TipcSrv methods:

Tut Method	TipcSrv Method	Purpose
`createOption`	Not available	Creates a custom option.
`getOption`	`getOption`	Returns a `TipcOption` object representing the requested option.
`getOptionBool`	`getOptionBool`	Returns the value of a `boolean` option.
`getOptionDouble`	`getOptionDouble`	Returns the value of a `double` option.
`getOptionInt`	`getOptionInt`	Returns the value of a `int` option.
`getOptionStr`	`getOptionStr`	Returns the value of a `string` option.
`loadOptionsFile`	`loadOptionsFile`	Loads the values of all options contained in the specified options database file.
`loadOptionsStream`	`loadOptionsStream`	Loads the values of all options contained in the specified InputStream.
`loadOptionsURL`	`loadOptionsURL`	Loads the values of all options contained in the options database located at the remote URL specified.
`removeOption`	Not available	Removes (and returns) a custom option.
`setOption`	`setOption`	Sets the value of an option, using the `TipcOption` helper class.

See the online documentation for the Tut, TipcOption, TipcConnClient, and TipcSrv classes for full usage details.

Setting Simple RTclient Options

RTclient options can be set to specific values by defining them in an option database and loading them, or by explicitly setting them. To set common options (that can be represented by a string) use the Tut.setOption convenience method or get the option and manipulate it with TipcOption's methods. An example of setting an option, in this case, ss.project, with the convenience method is:

```
Tut.setOption("ss.project", "myProject");
```

To accomplish this more formally with the TipcOption class, use this code:

```
TipcOption proj = Tut.getOption("ss.project");
proj.setValue("myProject");
```

Note that TipcOption objects are not to be directly instantiated by your code; they are created as necessary by Tut's createOption, getOption, and removeOption and returned to your application at that time.

See the online documentation for detailed information about the specific standard options recognized by the SmartSockets Java Class Library, as well as for valid values for all options.

Working with Enumerated Options

Often it is useful to have enumerated options, where the allowed values for an option are specified as a set of strings. The TipcOption class provides for enumerated options with automatic legal-value checking, as well as allowing the enumerated string values to be mapped onto integers. Individual enumerations may support integer mapping or simply strings, but not both. Mapped enumerations must have corresponding integer values provided for each of its string values; unmapped enumerations do not allow any corresponding integers to be specified.

This example creates an unmapped (simple) enumerated option, sets the legal values, and then sets and gets the option. Note that the second time getValueEnum() is called, an exception is thrown, because the value `"orange"` is not a legal enumeration value.

```
TipcOption enum = Tut.createOption("ss.col", "red");
enum.addEnumLegalValue("green");
enum.addEnumLegalValue("blue");
enum.setValue("green");
try {
   System.out.println("ss.col = " + enum.getValueEnum());
   enum.setValue("orange");
   System.out.println("ss.col = " + enum.getValueEnum())
}catch (TipcException te) {
   Tut.warning(te);
}
```

If you want to use a mapped enumeration, the code would look like:

```
TipcOption enum = Tut.createOption("ss.col", "red");
enum.addEnumMapLegalValue("red", 0);
enum.addEnumMapLegalValue("green", 1);
enum.addEnumMapLegalValue("blue", 2);
enum.setValue("green");
try {
   System.out.println("ss.col = " + enum.getValueEnumMap());
   enum.setValue("orange");
   System.out.println("ss.col = " + enum.getValueEnumMap());
}catch (TipcException te) {
   Tut.warning(te);
}
```

Loading RTclient Options from a File or URL

Option settings may be loaded from a local file or a URL, depending on security restrictions. Applets are generally not allowed access to the local file system, and typically retrieve options from a URL on the same web server from which the applet was downloaded.

To load options from a local file, use Tut.loadOptionsFile. Use Tut.loadOptionsURL to load options from a remote file using a URL. If your program has a different source for options that implements the InputStream interface, you can also use the Tut.loadOptionsStream method. All of these methods load all of the options immediately, overriding any existing settings for all of the options that appear in the file.

Options are also loaded from the System property table (the table returned by the standard Java library call System.getProperties). Option settings loaded from a file take precedence over settings that appear in the System table. All of the standard options have a default setting that is used if a setting does not appear in either the System table or a loaded file.

For example, examine this options file, `local.opt`:

```
ss.booleanValue1: true
ss.booleanValue2: false

ss.doubleValue1: 1.23456
ss.doubleValue2: 65432.1
ss.doubleValue3: -1.0
ss.doubleValue4: 100.0

ss.intValue1: 1
ss.intValue2: -16384
ss.intValue3: 2
ss.intValue4: -2
ss.intValue5: 32767
```

```
ss.bulb: on
ss.stringValue1: The quick red fox jumped over the lazy dog.
ss.stringValue2: Value1, value_2, VALUE-3, Value4, etc.
ss.charA: A
ss.charB: B
ss.charC: C
ss.charZ: Z

ss.project: foo_project
ss.server_names: _node, rocky, bullwinkle
ss.user_name: javauser
```

As you can see, the format is simply that of a Java property database, key-value pairs in ASCII, one pair to a line.

The following program, `getOptions.java` (see the "examples" directory on the distribution media) will load and interpret some of the values from such a file. Specifying a URL on the command line to `getOptions` allows loading of a file like the above from a remote location such as a web server.

The files for this lesson are located in the directories:

Windows

`%RTHOME%\java\tutorial\lesson5`

UNIX

`$RTHOME/java/tutorial/lesson5`

```
// getOptions.java
// Example of retrieving SmartSockets and user-defined options
// settings from a file, URL, or InputStream

import java.io.*;
import java.util.*;
import com.smartsockets.*;

public class getOptions {

  static public void main(String[] argv) {
     final String optionFile = "local.opt";

 // must create enumerated map option BEFORE loading!
  TipcOption bulb = null;
   try {
     bulb = Tut.createOption("ss.bulb", "off");
   }
   catch (TipcException te) {
     Tut.fatal(te);
   } // catch
   bulb.addEnumMapLegalValue("on", 1);
   bulb.addEnumMapLegalValue("off", 0);
   bulb.addEnumMapLegalValue("broken", -1);
```

```
    System.out.print("Getting options from ");

  // if argument was supplied, assume it's a URL to properties
    if (0 < argv.length) {
      System.out.println("URL " + argv[0]);
      Tut.loadOptionsURL(argv[0]);
    }
    else {
   // no argument, so use local options file
      System.out.println("local file " + optionFile);
      Tut.loadOptionsFile(optionFile);
   }  // else

   try {
     String project = Tut.getOptionStr("ss.project");
     System.out.println("ss.project = " + project);

     String server_names = Tut.getOptionStr("ss.server_names");
     System.out.println("ss.server_names = " + server_names);

     String user_name = Tut.getOptionStr("ss.user_name");
     System.out.println("ss.user_name = " + user_name);

  // get some user-defined options
  // these would be meaningful to this particular program
     boolean bv = Tut.getOptionBool("ss.booleanValue1");
     System.out.println("ss.booleanValue1 = " + bv);

     double dv = Tut.getOptionDouble("ss.doubleValue1");
     System.out.println("ss.doubleValue1 = " + dv);

     int iv = Tut.getOptionInt("ss.intValue1");
     System.out.println("ss.intValue1 = " + iv);

     String sv = Tut.getOptionStr("ss.stringValue1");
     System.out.println("ss.stringValue1 = " + sv);

      TipcOption so = Tut.getOption("ss.stringValue2");
      Vector ov = so.getValueList();
      if (null == ov) {
        System.out.println("Can't parse stringValue2!");
      }
      else {
    // use an Enumeration object to move through parsed list
        Enumeration en = ov.elements();
        for (int i=0; en.hasMoreElements(); i++) {
          String el = (String)en.nextElement();
         System.out.println("ss.stringValue2[" + i + "] = " + el);
       }  // for
     }   // else

     TipcOption sw = Tut.getOption("ss.bulb");
    System.out.println("ss.bulb(string) = " + sw.getValueEnum());
     int ev = sw.getValueEnumMap();
     System.out.println("ss.bulb(mapped) = " + ev);
    }
```

```
45    catch (TipcException te) {
46      Tut.warning(te);
      } // catch
    } // main
  } // getOptions
```

Let's examine the key lines of this program:

Lines 7-9 Create the mapped, enumerated option `bulb`; this should be done before options are loaded.

Lines 10-12 Enumerate the legal values and their mappings for the `bulb` option.

Lines 14-18 Load the options from a local file or the command-line specified URL.

Lines 19-32 Retrieve and display the values of various types of options.

Line 34 Gives an example of using getValueList to return a Vector of values (that were comma-separated in the property database file.)

Lines 41-44 Retrieve the `bulb` mapped enumerated option and display the string and mapped integer value.

Step 1 **Compile the getOptions.java program**

Compile the options program, `getOptions.java`, program using the command:

```
$ javac getOptions.java
```

Step 2 **Start the options program**

Make sure the `local.opt` file is present in the working directory, and start the options program using the command:

```
$ java getOptions
```

When you start the options program, this output is displayed:

```
Getting options from local file local.opt

ss.project = foo_project
ss.server_names = _node, rocky, bullwinkle
ss.user_name = javauser
ss.booleanValue1 = true
ss.doubleValue1 = 1.23456
ss.intValue1 = 1
ss.stringValue1 = The quick red fox jumped over the lazy dog.
ss.stringValue2[0] = Value1
ss.stringValue2[1] = value_2
ss.stringValue2[2] = VALUE-3
ss.stringValue2[3] = Value4
ss.stringValue2[4] = etc.
ss.bulb(string) = on
ss.bulb(mapped) = 1
```

Included with this example on the SmartSockets distribution media is an applet version, named `getOptionsApplet.java`. The applet version loads its options from the same web server (or applet viewer) it was downloaded from, but from the file `remote.opt`.

Making Custom Options Read-Only

For the custom options that you add, it is also possible to set the optional flag read-only. This will prevent the option from being reset during another `loadOptions` method or as part of your program. For example:

```
try {
 TipcOption ro = Tut.createOption("read-only", "false");
 ro.setValue("true");
 ro.setReadOnly(true);
 ro.setValue("changed");
}
catch (TipcException te) {
 Tut.warning(te);
}
```

The above code creates a new option, sets its value while it is still read-write, and then changes it to be a read-only value. When the code attempts to set the value a second time, an exception is thrown.

Java-Specific Options

There are several options only for Java RTclients. In Java, messages are read into the message queue by a reader thread. If the Java RTclient receives many large messages rapidly, it is possible for the Java Virtual Machine (JVM) to run out of memory. An alternative to increasing the Java heap size, the ss.max_read_queue_length or ss.max_read_queue_size can be modified to limit the number of messages read into the message queue at one time. The ss.max_read_queue_length limits the number of individual messages in the message queue and the ss.max_read_queue_size limits the total number of bytes in the message queue. A value of `0` indicates unlimited length or size. The ss.min_read_queue_percentage indicates to what point the message queue threshold must fall before messages are read into the message queue again. These values are changed only under unusual circumstances.

Summary

The key concepts covered in this lesson are:

- SmartSockets Java options are simply Java property database files containing keys that begin with ss. Option names and values are case sensitive.
- Many different options are recognized by the SmartSockets Java Class Library for controlling an RTclient's behavior, and you can add and destroy your own options with Tut.createOption and Tut.removeOption as well.
- Option (property) databases can be loaded from a file, a URL, or a currently open InputStream.
- When an option file is loaded, all values are loaded, overwriting any previous values in memory (except values marked "read-only").
- The Tut and TipcOption classes are used to manipulate SmartSockets Java options. TipcOption objects are not created by your code; they are created and returned to you as necessary by Tut methods.
- Simple option values can be set with the Tut.setOption convenience method and retrieved with the set of convenience methods in Tut, such as `getOptionBool`, `getOptionInt`, and so on.
- More complicated handling of options, such as setting the required or read-only flag or working with enumerated options, requires use of the TipcOption class.
- Enumerated options can have a set of legal values only or legal values and mapped integers, but the two types cannot be mixed within a single option.
- Legal values for enumerated options must be set before retrieving the option's value, or else an exception will be thrown.
- Values that contain comma-separated strings can be returned as a vector of strings with the TipcOption.getValueList method.
- Custom options can be made read-only with the TipcOption.setReadOnly method.
- The ss.max_read_queue_length limits the number of individual messages in the message queue and the ss.max_read_queue_size limits the total number of bytes in the message queue. The ss.min_read_queue_percentage indicates to what point the message queue threshold must fall before messages will begin to be read into the message queue again.

Chapter 8 Lesson 6: Java Applets

In this lesson you learn about:

- how to use SmartSockets from Java applets
- techniques for packaging SmartSockets with your applet for use over the web

Topics

Lesson 6 Overview

The files for this lesson are located in the directories:

Windows:
`%RTHOME%\java\tutorial\lesson6`

UNIX:
`$RTHOME/java/tutorial/lesson6`

This lesson illustrates techniques for using SmartSockets with Java applets, allowing publish-subscribe programs to be transferred over the web and executed in a client's web browser.

Applets and the Security Model

When developing applets that use SmartSockets (or any other type of networking) keep in mind the restrictions placed on applets by the Java Security Manager. Applets are only allowed to perform a subset of the tasks that a standalone Java application can, for reasons of security.

These restrictions may be configurable, based on the Java Virtual Machine (JVM) being used to execute your applet, but by default several key security restrictions are imposed that you should be aware of.

- Applets can only make network connections back to the machine from which they were downloaded
- Applets typically are not allowed to look up the name of the machine they are running on or its IP address
- Local file systems are usually completely off-limits to applets

These restrictions can have a significant impact on the design of your applet's use of SmartSockets.

Network Connections

Because a downloaded applet can only make connections back to the machine from which it was retrieved, use of the ss.server_names option is a must. Use ss.server_names to specify the machine (typically a web server) from which the applet was downloaded. This machine must be running RTserver or no connection is possible.

Because your applet is derived from the `Applet` class, there are useful methods that can be used to help facilitate the correct setting of ss.server_names. For example:

```
Tut.setOption "ss.server_names",this.getDocumentBase().getHost());
```

You might want to make a connection through a firewall, called tunneling through a firewall. Usually, this involves connecting to a proxy server. To do this, you need to use the HTTP_CONNECT proxy extension with your logical connection name. For example:

```
Tut.setOption ("ss.server_names",
"http_connect:www.company_name.com@tcp:server:5101)
```

See the *TIBCO SmartSockets User's Guide* for more information on tunneling through firewalls.

Local Machine Lookup

Typically, applets cannot determine the machine name nor IP address of the local machine. This type of information cannot be relied upon for RTclient applet identification purposes, such as setting unique subject name. Keep this in mind when designing SmartSockets applets.

Local File System Access

Do not design applets that require access to the local file system, for temporary storage or any other purpose, including file-based guaranteed message delivery (GMD). If you need to use GMD, use memory-based GMD. For more information, see Chapter 11, Guaranteed Message Delivery.

Remember, local files may not even be checked for existence.

Applet Life Cycle

Keep in mind that an applet's `init` method is usually invoked when the user presses the browser "reload" button; more fine-grained exception handling than usual may be necessary inside your `init` to ensure correct operation. Always test the various "start," "stop," "clone," and "reload" options in appletviewer to verify that your applet is working properly under all circumstances.

A good guideline is to register your callbacks in `init` and remove them in `destroy`. If you create message types in `init`, a "reload" will cause TipcSvc.createMt to throw an exception the second time through `init`, so be sure to plan for that appropriately.

Using Messaging Threads

To create effective SmartSockets applets, it is usually necessary to spin off a new messaging thread. This is simply a `run` method in some class that operates while the applet is allowed to execute. This thread loops, doing a TipcSrv.mainLoop call, allowing SmartSockets message processing to occur.

For example, the typical messaging thread infrastructure looks something like:

```
public class MyApplet extends Applet
  implements Runnable, ... {

    Thread reader = null;

  // messaging thread
    public void run() {
      Thread me = Thread.currentThread();
      me.setPriority(Thread.MIN_PRIORITY);
    while (reader == me) {
      try {
        TipcSrv srv = TipcSvc.getSrv();
        srv.mainLoop(1.0);
      } catch (TipcException te) { }
      try {
        Thread.sleep(100);
      } catch (InterruptedException ie) {
        break;
      } // catch
    } // messaging (reader) thread
  } // run
```

```
  public void stop() {
    reader = null;
  } // stop

  public void start() {
    reader = new Thread(this);
    reader.start();
  } // start

  .
  .
  .
} // class
```

The files for this lesson are located in the directories:

Windows:

`%RTHOME%\java\tutorial\lesson6`

UNIX:

`$RTHOME/java/tutorial/lesson6`

These files all contain code similar to the above. Note that changing the priority of the messaging thread is not necessary, but may help in some situations. Due to the scheduler inconsistencies between "green threads" and native threads, and native threads on differing platforms (Solaris versus Windows NT specifically), some experimentation may occasionally be necessary to achieve balanced messaging and CPU utilization.

Example Applet: ChatApplet

Presented below is an example SmartSockets applet. It implements a simple multi-user, real-time chat system using RTserver running on the machine from which the applet was downloaded. When first run, the applet prompts the user for a name. Once the name has been entered, the chat screen is displayed (a large text area for received messages and a small text field for entering messages). The source code is:

```
// ChatApplet.java
// Example applet: multiuser real-time chat

import java.util.Vector;
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import com.smartsockets.*;

public class ChatApplet extends Applet
implements TipcProcessCb, Runnable, ActionListener {

  static final String info = "Multiuser real-time chat applet
                                              demo.";
  static final int JAVA_CHAT_MT = 8081;
  static final String chat_subject = "/java_chat";

  TipcSrv srv;
  Thread reader = null;

  CardLayout cl = new CardLayout();
  TextArea out;
  TextField in, name;
  Button go_btn;
  Panel c1, c2;

  TipcMt chat_mt;    // message type
  TipcMsg chat_msg;  // message (will always be reused)
  TipcCb the_cb;

  Font my_font = new Font("Serif", Font.PLAIN, 11);
  String my_name;
```

```
public void run() {
  Thread me = Thread.currentThread();
  me.setPriority(Thread.MIN_PRIORITY);
  while (reader == me) {
    try {
      TipcSrv srv = TipcSvc.getSrv();
      srv.mainLoop(1.0);
    } catch (TipcException te) { }
    try {
      Thread.sleep(100);
    } catch (InterruptedException ie) {
      break;
    } // catch
  } // reader
} // run

public void init() {
  String host = getDocumentBase().getHost();
  try {
    if (!host.equals("")) {
      Tut.setOption("ss.server_names", host);
    } // if
  } catch (TipcException e) {}

  try {
    srv = TipcSvc.getSrv();
    srv.create();
  } catch (TipcException e) {}

  try {
    chat_mt = TipcSvc.createMt("java_chat", JAVA_CHAT_MT,
                               "str str");
  } catch (TipcException e) {
// after a reload, the create will throw an exception. since
// we want a handle to the mt, must look it up...
    chat_mt = TipcSvc.lookupMt(JAVA_CHAT_MT);
  } // catch

  chat_msg = TipcSvc.createMsg(chat_mt);
  chat_msg.setDest(chat_subject);

  try {
    srv.setSubjectSubscribe(chat_subject, true);
  } catch (TipcException e) {}

  the_cb = srv.addProcessCb(this, chat_mt, null);

  setupGUI();
} // init
```

```
public void destroy() {
  try {
    srv.removeProcessCb(the_cb);
    // user prop=-1 means to announce we've left the chat
    chat_msg.setUserProp(-1);
    announce(my_name, chat_subject);
  } catch (TipcException e) {}
} // destroy

void setupGUI() {
  setLayout(cl);
  c1 = new Panel();
  c1.add(new Label("Enter your name: "));
  name = new TextField("", 16);
  c1.add(name);
  go_btn = new Button("Go");
  go_btn.addActionListener(this);
  c1.add(go_btn);
  add("name", c1);

  c2 = new Panel(new BorderLayout());
  out = new TextArea("", 10, 60,
                  out.SCROLLBARS_VERTICAL_ONLY);
  out.setFont(my_font);
  out.setEditable(false);
  c2.add(new Label("Message Window"), "North");
  c2.add(out, "Center");
  Panel p = new Panel();
  p.add(new Label("Text to send:"));
  in = new TextField("", 60);
  in.setFont(my_font);
  p.add(in);
  Button do_chat = new Button("Send");
  do_chat.addActionListener(this);
  p.add(do_chat);
  c2.add(p, "South");
  add("chat", c2);

  cl.show(this, "name");
  name.requestFocus();
} // setupGUI
public void actionPerformed(ActionEvent ae) {
  if (ae.getActionCommand().equals("Go") &&
  name.getText().length()>0) {
    my_name = name.getText();
    cl.show(this, "chat");
    in.requestFocus();
```

```
    // request replies to our announcement with user prop=1
        chat_msg.setUserProp(1);
        announce(my_name, chat_subject);
      } // if it's the go button
      if (ae.getActionCommand().equals("Send") &&
        in.getText().length()>0) {
        try {
          chat_msg.setNumFields(0);
          chat_msg.appendStr(my_name);
          chat_msg.appendStr(in.getText());
          srv.send(chat_msg);
          srv.flush();
        }
        catch (TipcException te) {
          Tut.warning(te);
        } // catch
        in.setText("");
        in.requestFocus();
      } // if it's the send button
    } // actionPerformed

    public String getAppletInfo() {
      return info;
    } // getAppletInfo

    public void stop() {
      reader = null;
    } // stop

    public void start() {
      reader = new Thread(this);
      reader.start();
      repaint();
    } // start

    void announce(String name, String dest) {
     // let the new person know we're here or announce ourselves
    try {
        chat_msg.setDest(dest);
        chat_msg.setNumFields(0);
        chat_msg.appendStr(my_name);
        chat_msg.appendStr("");
        srv.send(chat_msg);
        srv.flush();
        chat_msg.setDest(chat_subject);
     }
     catch (TipcException e) {
        Tut.warning(e);
     } // catch
    } // announce

    public void process(TipcMsg msg, Object o) {
      try {
        msg.setCurrent(0);
        String who = msg.nextStr();
        String text = msg.nextStr();
```

```
        // see if it's an announcement
124         if (text.equals("")) {
125           if (msg.getUserProp() == -1) {
126              out.append("* Leaving chat: " + who + " *");
              else {
127             out.append("* Joining chat: " + who + " *");
              }
128           if (!who.equals(my_name) && msg.getUserProp() == 1) {
129              chat_msg.setUserProp(0);
130              announce(who, msg.getSender());
              }  // if "wants replies" flag is set and not our message
            }  // if an announcement
            else {
          // a 'regular' message; display it
131            out.append("[" + who + "] " + text);
            }  // if
132       } catch (TipcException e) {
133         Tut.warning(e);
          }  // catch
        }  // process

      }  // main class
```

Let's examine some of the key parts of the applet:

Line 12 — Declares the class-level variable `reader`, which will be used to control the messaging thread.

Lines 23-32 — Implement the messaging thread. Notice that the thread processes messages for one second (line 28) and then sleeps for 0.1 second (line 30) repeatedly.

Lines 33-38 — Sets the ss.server_names option to point back to the server we downloaded from, because the default applet security does not allow us to connect to a local RTserver. In many applet situations it's very likely there is no RTserver available except on the machine from which the applet was downloaded.

Lines 45-46 — Creates the template message for chatting that will be reused during the life of this process.

Lines 51-55 — Provides the `destroy` method, called when the applet is reloaded or shut down. Unregisters our message processing callback and sends out a final "leaving chat" message, via `announce`. The receiving chat clients will know this is a parting announcement by checking the message's user property, which is set here to `-1`.

Lines 56-83 — Build the applet's GUI. Two panels are used, one containing the login screen, the other for the chat and input window. They are arranged with a `CardLayout`, the login screen displayed "on top" first.

Lines 84-91	If the button on the login screen is pressed, the user name is recorded for inclusion in future outgoing messages (variable *my_name*), and an announce message is sent with the user property set to `1`. The user property is used in this instance to request that other chat clients reply directly back to this client after the announcement, announcing themselves. In this way a new addition to the chat "room" will receive an initial round of announcements from all other clients currently present.
Lines 92-101	When the "send message" button is pressed, this code first clears the message's old contents (line 93). It then builds the SmartSockets message to be published, publishes it, and flushes the server connection to ensure timely delivery.
Line 105	When an applet's `stop` method is called, it is desirable to disable the messaging thread, so the `reader` variable is invalidated.
Lines 107-108	Upon starting an applet, the messaging thread is created and started.
Lines 110-119	The announce method is used to send a SmartSockets chatting message announcing one of three events: a client joining the conversation, a client leaving the conversation, or this client's presence in response to a new client's entry. Each type of announcement contains our name as the first string field and a blank string as the second field.
Line 121	Sets the message's current field pointer to `0`, the first field, to prepare the message for field extraction.
Lines 125-127	Display the appropriate message for clients joining / departing the discussion.
Lines 128-130	If the announcement was to declare a client joining the conversation, requesting a reply from others in the "room" (and it wasn't generated by this client), send a reply announcement directly to the requesting client.
Line 131	Displays the contents of a general chat message sent by a client (including this client).

As with all applets, we need to create an HTML file to facilitate its download and launch. This is a listing of `ChatApplet.html` (note that the listing is a "standard" applet invocation; for use with the Java Plug-In, additional tags are required):

```
1 <html>
2 <head><title>SmartSockets Chat Applet</title></head>
3 <body>
4   <h1>SmartSockets Chat Applet</h1><hr>
5   <applet
6   code="ChatApplet.class"
7   width=600 height=400
8   archive="ss.jar">
9     (SmartSockets Applet)
10  </applet>
11  </body>
12 </html>
```

Items to note about the above HTML are:

Line 5 Begins the `<applet>` tag.

Line 6 Specifies the class that will be run by this applet.

Line 7 Specifies the on-screen size of the applet.

Line 8 Indicates that the ss.jar SmartSockets archive is to be "preloaded". Note that this file must be in the same directory as the compiled `.class` file, whether using appletviewer or viewing via a web browser.

If you do not wish to copy the ss.jar file during testing with appletviewer, simply leave off the `archive` modifier until these files are placed on a web server.

Your CLASSPATH setting should allow appletviewer to locate the SmartSockets archive.

Line 9 Alternate text, to be displayed in browsers that do not support Java, but do understand the `<applet>` tag.

To see the applet in action, use this procedure:

Step 1 **Ensure RTserver is running**

Make sure RTserver is running. If not, start it:

```
$ rtserver
```

On platforms that support both 32- and 64-bit, use the `rtserver64` command to run the 64-bit version of the `rtserver` script.

Step 2 **Compile the ChatApplet.java program**

Compile the `ChatApplet.java` program:

```
$ javac ChatApplet.java
```

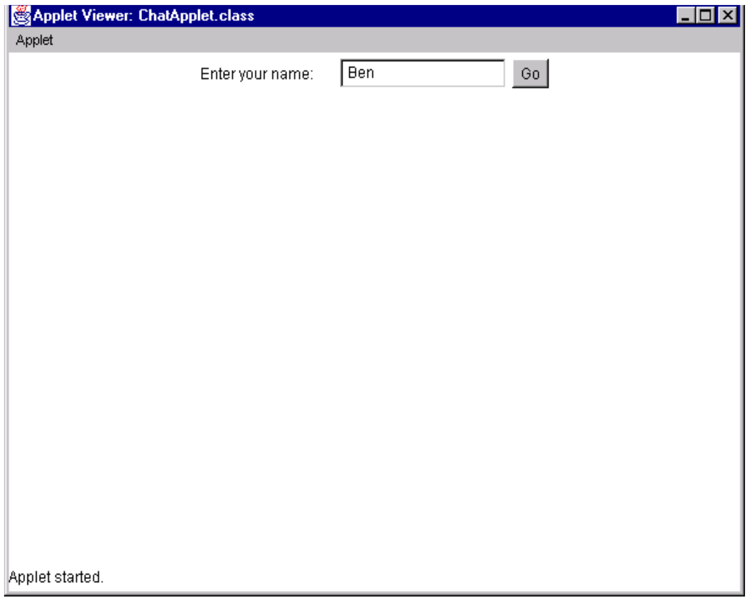
Step 3 **Start the chat applet program**

Start the chat applet:

```
$ appletviewer ChatApplet.html
```

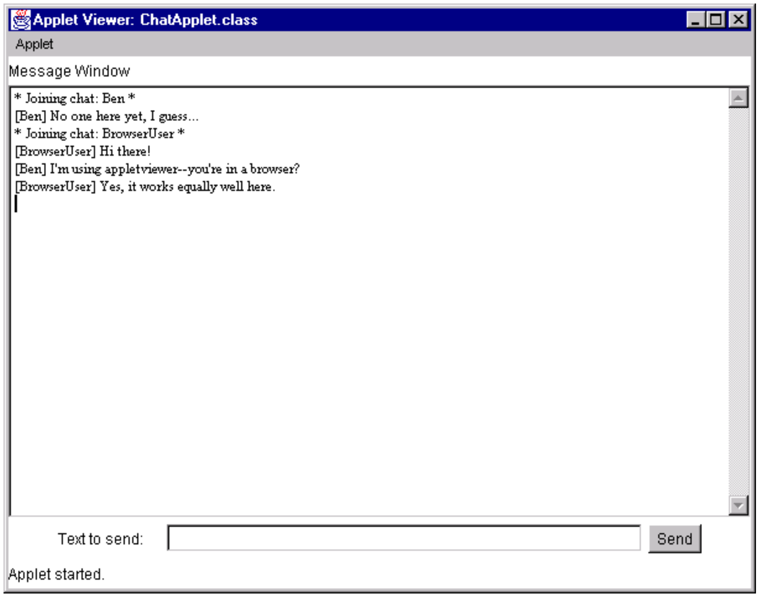
The applet's window should first display the log-in screen as shown in Figure 7.

Figure 7 Applet Viewer display of ChatApplet (login phase)



Once the user is logged in and a conversation is started, the view from appletviewer might look like Figure 8 below:

Figure 8 Applet Viewer display of ChatApplet (chat phase)



The other side of the conversation (in this case there are only two chatters, but there could be very many) might look like Figure 9 if the user were using a web browser instead of an appletviewer.

Figure 9 Browser display of ChatApplet

As you can see, the two users are carrying on a conversation using SmartSockets messaging. The web server at which the browser is pointed, `trojan`, happens to be the same machine on which appletviewer is running. This does not need to be the case, however; as long as `trojan` has RTserver running (for the applet to connect back to) and the appletviewer instance connects to an RTserver process in the same server cloud as `trojan`'s, messages will be successfully delivered.

Summary

The key concepts covered in this lesson are:

- When building applets with the SmartSockets Java Class Library (and networking applets in general), keep in mind the security restrictions placed on applets, such as limited host connectivity and lookup and little or no local file access.
- When deploying SmartSockets applets with typical security restrictions, RTserver must be running on the same host as the web server for downloaded applets to connect back to.
- Applets can have very different life cycles from applications; care must be taken to ensure that your program will respond and survive "reloads" and other actions. Sun’s appletviewer is a very good tool for testing these cases.
- SmartSockets applets should utilize a messaging thread, spun off by your applet’s `start` method. Performing a TipcSrv.mainLoop to ensure message delivery should be this thread’s sole responsibility.
- Not all Java Virtual Machines (JVMs) especially those in web browsers, necessarily behave identically. We recommend testing applets with Sun Microsystems JSDK and the appletviewer utility or with the HotJava browser.

Congratulations!

You have successfully completed the SmartSockets Java class library tutorial! For more information on the Java class library, see the online SmartSockets Java class reference information in Javadoc format in the `c:`*install_dir*`\ss68\doc` directory.

For more information on the RTclient options available in Java, see Chapter 9, RTclient Options.

For more information on using GMD with Java, see Chapter 11, Guaranteed Message Delivery.

Chapter 9 RTclient Options

This chapter describes RTclient option databases and the options available in those database files.

All of the option handling methods are in the Tut utility class. See the SmartSockets online Java class reference information, provided in Javadoc format, for the Tut class for details about specific methods. This chapter provides an introduction to the SmartSockets option system, demonstrates the basics of setting and retrieving options, and defines the available standard options.

Topics

Option (Property) Databases

SmartSockets options are retrieved from Java Properties databases. These databases take the form of text files containing key-value pairs, one entry per line. The "key" string may contain periods (.) to indicate property hierarchies, and all SmartSockets options reside in the "ss." hierarchy. For example, two of the standard SmartSockets options properties are defined in the property database:

```
ss.server_auto_connect=false
ss.project=foo
```

Note that option names and values are case-sensitive and can be presented in any order. Options properties should appear only once in the property database. If multiple instances of an option appear with conflicting properties, the last key-value pair in the database is used, overriding any earlier settings. Option databases can contain values for the standard RTclient options, and user-defined settings.

The options follow standard Java property conventions. For example, you can use a pound sign (#) to comment out a line in a file.

Loading RTclient Options

Option settings may be loaded from a local file or a URL, depending on security restrictions. Applets are generally not allowed access to the local file system and typically retrieve options from a URL on the same web server from which the options were downloaded.

To load options from a local file, use Tut.loadOptionsFile. Use Tut.loadOptionsURL to load options from a remote file using a URL. Both methods load the options immediately, overriding any existing settings for the options that appear in the file.

Options are also loaded from the System property table (the table returned by the standard Java library call `System.getProperties`). Option settings loaded from a file take precedence over settings that appear in the System table. All of the standard options have a default setting that will be used if a setting does not appear in either the System table or a loaded file.

Setting RTclient Options

RTclient options can be set to specific values by defining them in an option database and loading them or by using the Tut.setOption method.

When entering multiple values for list options, the values must be separated by commas. Options are case sensitive in Java.

Table 3 lists and briefly describes the relevant options available for Java RTclients. For complete descriptions of these options, see *TIBCO SmartSockets User's Guide*.

Table 3 Java RTclient Options

Option Name	Type	Default
ss.backup_name	string	~
ss.compression	boolean	`false`
ss.compression_args	integer	`6`
ss.compression_name	string	none
ss.compression_stats	boolean	`false`
ss.default_msg_priority	numeric	`0` (zero)
ss.default_protocols	string list	`tcp`
ss.default_subject_prefix	string	none
ss.enable_control_msgs	string list	`quit`
ss.group_names	string	`rtworks`
ss.ipc_gmd_directory	string	the directory name where Java is installed as defined by the Java property `java.home` or "." if `java.home` is not set
ss.ipc_gmd_type	string	`default`
ss.log_in_data	string	none
ss.log_in_internal	string	none
ss.log_in_status	string	none

Table 3 Java RTclient Options

Option Name	Type	Default
ss.log_out_data	string	none
ss.log_out_internal	string	none
ss.log_out_status	string	none
ss.max_read_queue_length	numeric	0 messages
ss.max_read_queue_size	numeric	500000 bytes
ss.mcast_cm_file	string	none
ss.min_read_queue_percentage	numeric	50 percent
ss.monitor_ident	string	RTclient
ss.monitor_level	standard	RTclient
ss.monitor_scope	string	/*
ss.project	identifier	rtworks
ss.proxy.password	string	none
ss.proxy.username	string	none
ss.server_auto_connect	boolean	true
ss.server_auto_flush_size	numeric	32768
ss.server_delivery_timeout	numeric	30.0 seconds
ss.server_disconnect_mode	identifier	gmd_failure
ss.server_keep_alive_timeout	numeric	15.0 seconds
ss.server_max_reconnect_delay	numeric	30.0
ss.server_msg_send	boolean	true
ss.server_names	string list	*_node*
ss.server_read_timeout	numeric	30.0 seconds
ss.server_start_delay	numeric	1.0 seconds

Table 3 Java RTclient Options

Option Name	Type	Default
ss.server_start_max_tries	numeric	`1`
ss.server_write_timeout	numeric	`30.0` seconds
ss.socket_connect_timeout	numeric	`5.0`
ss.subjects	string list	none
ss.time_format	identifier	`unknown`
ss.trace_file	string	`unknown`
ss.trace_file_size	numeric	`0`
ss.trace_flags	string list	`prefix`
ss.trace_level	string	`unknown`
ss.unique_subject	string	*_node_start-time*
ss.user_name	string	*username* if available; `rtworks` otherwise

ss.backup_name

Type: String

Default Value: UNIX and Windows: ~

OpenVMS: None

Valid Values: Any valid filename characters

The ss.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 ss.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.

ss.compression

Type: Boolean

Default Value: false

Valid Values: true or false

The ss.compression option specifies whether connection-level compression is enabled. If set to true then all data sent on all connections is compressed. The actual compression algorithm used is specified by the ss.compression_name.

ss.compression_args

Type: Integer

Default Value: `6`

Valid Values: Valid arguments for the library specified by ss.compression_name.

The ss.compression_args option allows arguments specific to the compression library specified by the ss.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.

ss.compression_name

Type: String

Default Value: none

Valid Values: Any valid SmartSockets compression Library.

The ss.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.

ss.compression_stats

Type: Boolean

Default Value: `false`

Valid Values: `true` or `false`

The ss.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.

ss.default_msg_priority

Type: Integer

Default Value: 0

Valid Values: Any integer between -32768 and 32767, inclusive

The ss.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 the value you specify for ss.default_msg_priority (if the message type priority is unknown).

ss.default_protocols

Type: String List

Default Value: tcp

Valid Values: Any valid Java protocol

The ss.default_protocols option specifies a list of IPC protocols to try if no protocol is listed in a logical connection name in the ss.server_names option.

ss.default_subject_prefix

Type: String

Default Value: unknown

Valid Values: Any string beginning with a backslash (/)

The ss.default_subject_prefix option specifies the qualifier to prepend to message subject names that do not start with a slash (/). Subject names are organized in a hierarchical namespace where the components are delimited by a slash. A subject name that starts with a slash is called an absolute subject name. All subject names that do not begin with a slash have the value you specify for ss.default_subject_prefix prepended to them to create a fully qualified name for the hierarchical subject namespace.

If the option is unset (unknown), RTclient uses the RTserver Default_Subject_Prefix value. For more information, see the *TIBCO SmartSockets User's Guide*.

ss.enable_control_msgs

Type: String List

Default Value: `quit`

Valid Values: `quit` or `_all` or `unknown`

The ss.enable_control_msgs option is a list specifying the commands allowed in a CONTROL message. The default allows the inclusion of the `quit` command using a CONTROL message. When this option is set to `unknown`, all commands (including `quit`) are disabled. Setting this option to `_all` allows the inclusion of all valid commands in a CONTROL message. `quit` is the only valid command for Java.

ss.group_names

Type: String List

Default Value: `rtworks`

Valid Values: Any valid multicast group name

The ss.group_names option specifies a list of multicast groups. 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.

This option is optional, and is only used if you have a license for the SmartSockets Multicast option.

ss.ipc_gmd_directory

Type: String

Default Value: The directory name where Java is installed as defined by the Java property `java.home` or "." if `java.home` is not set

Valid Values: Any valid directory name

The ss.ipc_gmd_directory option is only for use with guaranteed message delivery (GMD).

This option specifies the location for the GMD files. The default location for GMD files is the directory where Java is installed, as obtained using `System.getProperty("java.home")`. You can specify a different directory if you like.

The GMD directory is the base directory under which a directory named `gmd` is created. Under that, a directory containing the unique subject is created. If these subdirectories do not already exist, SmartSockets Java creates them.

ss.ipc_gmd_type

Type: String

Default Value: `default`

Valid Values: `default` or `memory`

The ss.ipc_gmd_type option is only for use with GMD (guaranteed message delivery).

This option 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, even if ss.unique_subject is set.

ss.log_in_data

Type: String

Default Value: None

Valid Values: Any valid file name

The ss.log_in_data option specifies the name of the file that RTclient 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 the *TIBCO SmartSockets User's Guide*.

ss.log_in_internal

Type: String

Default Value: None

Valid Values: Any valid file name

The ss.log_in_internal option specifies the name of a file that RTclient uses to log incoming internal messages, such as MON_SUBJECT_SUBSCRIBE_SET_WATCH or CONNECT_CALL, that are received from RTserver. If this option is not set, incoming internal messages are not logged. Internal messages are listed in the *TIBCO SmartSockets User's Guide*.

ss.log_in_status

Type: String

Default Value: None

Valid Values: Any valid file name

The ss.log_in_status option specifies the name of a file that RTclient 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 the *TIBCO SmartSockets User's Guide*.

ss.log_out_data

Type: String

Default Value: None

Valid Values: Any valid file name

The ss.log_out_data option specifies the name of a file that RTclient 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 the *TIBCO SmartSockets User's Guide*.

ss.log_out_internal

Type: String

Default Value: None

Valid Values: Any valid file name

The ss.log_out_internal option specifies the name of a file that RTclient uses to log outgoing internal messages, such as MON_SUBJECT_SUBSCRIBE_SET_WATCH or CONNECT_CALL, that are sent to RTserver. If this option is not set, outgoing internal messages are not logged. Internal messages are listed in the *TIBCO SmartSockets User's Guide*.

ss.log_out_status

Type: String

Default Value: None

Valid Values: Any valid file name

The ss.log_out_status option specifies the name of a file that RTclient 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 the *TIBCO SmartSockets User's Guide*.

ss.max_read_queue_length

Type: Integer

Default Value: 0

Valid Values: Any integer 0 or greater

The ss.max_read_queue_length option specifies the number of individual messages allowed in the message queue. After the value is reached, subsequent messages are buffered in the RTserver until the queue length falls below a message queue threshold. This threshold is calculated as: (the value of ss.max_read_queue_length) multiplied by (the value of ss.min_read_queue_percentage).

The default setting, 0, allows an unlimited or infinite number of messages in the queue. This is the recommended setting. However, it is possible that if your Java RTclient receives many large messages rapidly, the JVM can run out of memory. Instead of increasing the Java heap size, you can limit the number of messages allowed in the queue by changing this option from 0 to another number. Related options are ss.max_read_queue_size and ss.min_read_queue_percentage. For additional information, see Java-Specific Options on page 117.

ss.max_read_queue_size

Type: Integer

Default Value: 500000

Valid Values: Any integer 0 or greater

The ss.max_read_queue_size option specifies the limit for the total number of bytes allowed in the message queue. After the value is reached, subsequent messages are buffered in the RTserver until the queue size falls below a message queue threshold. This threshold is calculated as: (the value of ss.max_read_queue_length) multiplied by (the value of ss.min_read_queue_percentage). The default setting, 500000, is the recommended setting. However, it is possible that if your Java RTclient receives many large messages rapidly, the JVM can run out of memory. Instead of increasing the Java heap size, you can limit the size of the message queue by changing this option.

Setting this option to a value of `0` disables the option, allowing an unlimited (infinite) queue size and no limit to the number of messages in the queue.

Related options are ss.max_read_queue_length and ss.min_read_queue_percentage.

ss.mcast_cm_file

Type: String

Default Value: None

Valid Values: Any valid pathname, specified without `%` characters

The ss.mcast_cm_file option specifies the fully qualified pathname to the `mcast.cm` file the RTclient process should use. For more information on the `mcast.cm` file and it's use see the section on PGM options in the *TIBCO SmartSockets User's Guide.*

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%`.

ss.min_read_queue_percentage

Type: Integer

Default Value: `.5`

Valid Values: Any integer from 0 to 1.0, inclusive

The ss.min_read_queue_percentage option specifies the percentage to which the message queue threshold must fall before any additional messages are read into the queue. The default value, `.5`, is the recommended setting. Changing this setting is only recommended if you are having problems with your JVM running out of memory. Related options are ss.max_read_queue_length and ss.max_read_queue_size.

ss.monitor_ident

Type: String

Default Value: `RTclient`

Valid Values: Any valid string

The ss.monitor_ident option specifies an identifying string for an RTclient. This identifier is used as a descriptive name for the RTclient when it is being monitored by SmartSockets or by another RTclient that is monitoring RTclient extension data. See the *TIBCO SmartSockets User's Guide* for information on monitoring.

The string is sent to the RTserver when the RTclient connects to the RTserver. An RTclient that sets this option after connecting to an RTserver is not identified properly for monitoring purposes.

ss.monitor_ident is required when monitoring an RTclient. It cannot be unset after it is set.

ss.monitor_level

Used for: RTclient, RTmon, and RTgms processes

Type: String

Default Value: standard

Valid Values:

- none -- not implemented in this release
- standard -- all monitoring except traffic monitoring
- all -- all monitoring, including traffic monitoring and memory or cpu intensive monitoring

The ss.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. Unless ss.monitor_level is set to `all`, the RTclient will not respond to poll calls by other RTclients.

This option is required and cannot be unset.

ss.monitor_scope

Type: String

Default Value: `/*`

Valid Values: Any valid subject name character or characters

The ss.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.

ss.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 ss.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 *TIBCO SmartSockets User's Guide*.

ss.project

Type: String

Default Value: `rtworks`

Valid Values: Any valid project name

The ss.project option specifies the name of the SmartSockets project to which the RTclient is connected. The RTclient only communicates with other SmartSockets processes that have the same project name.

ss.proxy.password

Type: String

Default Value: None

Valid Values: Any valid password

The ss.proxy.password option is a string that specifies the user password that the RTclient provides to a proxy server for authentication. This option is only used when connecting to a proxy server that requires authorization. If both ss.proxy.username and ss.proxy.password are set, they are sent to the proxy server as part of authentication.

ss.proxy.username

Type: String

Default Value: None

Valid Values: Any valid username

The ss.proxy.username option is a string that specifies the username that the RTclient provides to a proxy server for authentication. This option is only used when connecting to a proxy server that requires authorization. If both ss.proxy.username and ss.proxy.password are set, they are sent to the proxy server as part of authentication.

ss.server_auto_connect

Type: Boolean

Default Value: `true`

Valid Values: `true` or `false`

The ss.server_auto_connect option specifies whether or not the RTclient should automatically create a connection to RTserver if one is needed, such as when a read or write operation is attempted before a connection is established. If ss.server_auto_connect is set to `false`, then the RTclient does not attempt to recreate a connection to the RTserver automatically, and outgoing messages are simply buffered.

ss.server_auto_flush_size

Type: Integer

Default Value: `32768`

Valid Values: Any integer 0 or greater

The ss.server_auto_flush_size option specifies the size, in bytes, that the outbound buffer can reach before the data is automatically flushed (that is, written to the connection). If ss.server_auto_flush_size is set to `0`, all outgoing messages are automatically flushed immediately. If the RTclient is sending many messages in a short period of time, setting ss.server_auto_flush_size to a larger value can lessen the amount of CPU time that the RTclient uses.

ss.server_delivery_timeout

Type: Real Number

Default Value: `30.0`

Valid Values: Any real number 0.0 or greater

The ss.server_delivery_timeout option specifies the 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. When this option is set to `0.0`, then this option is disabled, and the sending process never times out.

When a guaranteed message is sent to RTserver but is not acknowledged within the specified period of time, then an error has occurred, and a GMD_FAILURE message is processed.

ss.server_disconnect_mode

Type: String

Default Value: `gmd_failure`

Valid Values: `warm`, `gmd_failure`, or `gmd_success`

The ss.server_disconnect_mode option specifies what the RTserver should do, if anything, if and when an RTclient disconnects from RTserver. The three possible values are:

`warm`	specifies that RTserver saves subject, project, and guaranteed message delivery information about the disconnecting RTclient so that no messages are lost.
`gmd_failure`	specifies that RTserver destroys all information about the disconnecting RTclient and causes pending guaranteed message delivery to fail.
`gmd_success`	specifies that RTserver destroys all information about the disconnecting RTclient, but causes pending guaranteed message delivery to succeed.

Setting the option to `warm` is useful when an RTclient must run continuously and not lose any messages even if the RTclient crashes or the connection is accidentally terminated. In this mode, RTserver remembers the subjects being subscribed to by the disconnecting RTclient and buffers guaranteed messages. When an RTclient with the same subject name (specified using the ss.unique_subject option) reconnects to RTserver, it resends the guaranteed messages (that were saved in the buffer) to that RTclient. The RTserver option Client_Reconnect_Timeout controls the maximum amount of time, in seconds, that RTserver waits (and saves the information) for a disconnected RTclient to reconnect. If the RTclient does not reconnect to RTserver within the specified period of time, then RTserver clears the unacknowledged messages and sends a GMD_NACK message back to the sender of those 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 the disconnected RTclient process and sends a GMD_NACK message back to the sender of those messages.

Setting the option to `gmd_success` is useful for short-lived operations or when an RTclient wants to exit cleanly without causing GMD failure in the sending process. In this mode, RTserver clears the guaranteed messages that have not been acknowledged by the disconnecting RTclient, but still sends an acknowledgment of delivery (a GMD_ACK message) back to the sender of those messages.

ss.server_keep_alive_timeout

Type: Real Number

Default Value: `15.0`

Valid Values: Any real number 0.0 or greater

The ss.server_keep_alive_timeout option specifies how long, in seconds, the RTclient waits for RTserver to respond to a keep alive query. This check is called a keep alive. When this option is set to 0.0, keep alives are disabled. If the keep alive fails, then the error callback is triggered with a timeout error. The larger the value specified, the longer the RTclient waits to detect a possible RTserver failure. If this value is set too low, however, the RTclient may mistakenly think that it has lost its connection to RTserver when RTserver is merely busy.

ss.server_max_reconnect_delay

Type: Real Number

Default Value: `30.0`

Valid Values: Any real number 0.0 or greater

The ss.server_max_reconnect_delay option specifies the upper bound on a random delay introduced when an RTclient has to reconnect to RTserver.

This option is useful when an RTserver with many clients fails and all of those RTclients are attempting to reconnect. The delay enhances total reconnect time by slightly staggering reconnect requests. Setting the option to zero disables the delay.

ss.server_msg_send

Type: Boolean

Default Value: `true`

Valid Values: `true` or `false`

The ss.server_msg_send option specifies whether or not an RTclient 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 ss.server_msg_send. This option is useful for backup processes that should receive messages from RTserver but not send any out.

ss.server_names

Type: String List

Default Value: `_node`

Valid Values: Any valid logical connection names, separated by commas

The ss.server_names option specifies a list of logical connection names used to find an RTserver. The RTserver names should be listed in order of preference, separated by commas. If the connection to RTserver is lost, then a connection is established with the next RTserver listed, in a circular fashion, until an RTserver responds and a stable connection is established. Each logical connection name has the form *protocol:nodename:port*. You can delete the protocol (which must be `tcp` for Java) and the port, in which case the default is used. Only *nodename* is required. The string "`_node`" can be used in place of the local nodename.

ss.server_read_timeout

Type: Real Number

Default Value: `30.0`

Valid Values: Any real number 0.0 or greater

The ss.server_read_timeout option specifies how long, in seconds, an RTclient waits for traffic on the connection before it issues a keep alive request. This timeout is used to check for possible network failures. When this option is set to zero (`0.0`), then read timeouts are disabled. The larger the value set, the longer the RTclient waits to detect a possible RTserver failure.

ss.server_start_delay

Type: Real Number

Default Value: `1.0`

Valid Values: Any real number 0.0 or greater

The ss.server_start_delay option specifies how long, in seconds, that an RTclient sleeps between traversals of each RTserver (connection names) as specified in the ss.server_names option.

Unlike the C API, the Java API does not attempt to start a new RTserver. This option merely specifies how long to wait between connection attempts.

ss.server_start_max_tries

Type: Integer

Default Value: `1`

Valid Values: Any integer 1 or greater

The ss.server_start_max_tries option specifies how many times to traverse the list specified by the ss.server_names option attempting to find an RTserver. RTclient is not able to communicate with other SmartSockets processes if it cannot create a connection to RTserver.

ss.server_write_timeout

Type: Real Number

Default Value: `30.0`

Valid Values: Any real number 0.0 or greater

The ss.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, the error callback is triggered with a timeout error. When this option is set to zero (`0.0`), then write timeouts are disabled. The larger the value set, the longer the RTclient must wait to detect a possible RTserver failure. If this is set too low, however, the RTclient may mistakenly think that it has lost its connection to RTserver when RTserver is merely very busy.

ss.socket_connect_timeout

Type: Real Number

Default Value: `5.0`

Valid Values: Any real number 0.0 or greater

The ss.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 ss.socket_connect_timeout is set to `0.0`. All SmartSockets standard processes use sockets for inter-process communication.

Proper operation requires JRE 1.4 or later.

ss.subjects

Type: String List

Default Value: None

Valid Values: Any valid subject names, separated by commas

The ss.subjects option specifies a list of the initial subjects to which the RTclient is to subscribe. Multiple subjects can be listed and must be separated by commas:

```
ss.subjects: /system/eps, /system/pcs, /control/...
```

In addition to this list, the subject named in the ss.unique_subject option is automatically subscribed to when the RTclient connects to RTserver.

The RTclient can subscribe to subjects at any time using the `subscribe` command.

ss.time_format

Type: String (Identifier)

Default Value:
- `hms` for RTserver and RTmon processes
- `unknown` for all other RT processes

Valid Values:
- `full`
- `fullzone`
- `hms`
- `numeric`

The ss.time_format option controls how the RT process displays the value of time. Three standard formats and one user-defined format are available:

`full` displays a combination of the date and the time.

`fullzone` displays a combination of the date and the time with a time zone designation

`hms` displays the time in hours, minutes, and seconds.

`numeric` displays the floating point representation of time.

ss.trace_flags

Type: String List (Identifiers)

Default Value: `prefix`

Valid Values: `prefix`, `timestamp`, `unknown`

The ss.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
```

ss.unique_subject

Type: String

Default Value: *_node_start-time*

Valid Values: Any unique value

The ss.unique_subject option specifies a unique subject that the RTclient automatically subscribes to when it creates a connection to RTserver. RTserver does not allow multiple processes in the same project to have the same unique subject name. The default value is *_node_start-time*, where:

node is the network node name of the computer on which RTclient is running.

start-time is the time the RTclient started, expressed as a hexadecimal value of milliseconds.

ss.user_name

Type: String

Default Value: *username* if found in the current Java environment

`rtworks` if *username* is not available

Valid Values: Any valid user name

The ss.user_name option describes the user that launched the RTclient. Applets in some restricted environments may not have access to the system user name property. This option is set with the correct user name if it could be retrieved from the Java environment. If it could not, it is set to `rtworks`. Java RTclients can override the value with a more appropriate value.

Chapter 10 Using Java Clients

The basic information on SmartSockets clients is contained in the *TIBCO SmartSockets User's Guide*. Many of the features and much of the usage and client interaction is similar for Java and C clients. This chapter discusses areas where Java is unique that were not covered in the tutorials. For more information on Java and guaranteed message delivery, see Chapter 11, Guaranteed Message Delivery. For information on how the Java APIs map to the C APIs, see Appendix A, Java API to C API Mapping.

Topics

Using TIBCO SmartSockets Multicast

In addition to standard publish-subscribe with RTserver and RTclient, SmartSockets provides a multicast option 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 option enables messages to be multicast to RTclients. The SmartSockets Multicast option uses the PGM protocol to route messages and a new RT process called RTgms to handle the message routing. There are special PGM 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 a SmartSockets Multicast license, separate from your standard SmartSockets license. The SmartSockets Multicast option is available on UNIX and Windows platforms. Contact your TIBCO sales representative for more information. 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 or higher runtime libraries. To use the multicast protocol, PGM, your network hardware, such as routers and switches, must configured for multicast use.

For more information on multicasting or working with RTgms, see the *TIBCO SmartSockets User's Guide*. The rest of this section covers Java-related information only.

Using Multicast with Java

To use multicast with Java, there are several things you must do:

1. Configure and start an RTgms process to manage the multicast routing. There is no Java version of RTgms--like RTservers, it is strictly C-based. For information on configuring, starting, and managing RTgms processes, see the chapter on multicast in the *TIBCO SmartSockets User's Guide*.

2. Use the default configuration for multicast or create the two multicast command files, `mcast.cm` and `mcastopts.cm`, for your RTclients that want to use multicast.

   The `mcast.cm` command file uses special PGM options to control the PGM aspects of multicast, and these options are the same in both C and Java. These options must be set in the `mcast.cm` file.

   The `mcastopts.cm` command file contains one option, mcast_cm_file, which you can use to specify the location of your `mcast.cm` file if you do not want to use the default location or you want to share a single `mcast.cm` file across multiple systems. The value you specify should be a fully qualified pathname.

   For information on the multicast command files and setting the PGM options, see the chapter on multicast in the *TIBCO SmartSockets User's Guide*.

3. Add `ss-pgm.jar` to your CLASSPATH environment variable. See your operating system documentation for instructions to add a file name to the CLASSPATH.

4. Create a connection to the RTgms process from your RTclient. The RTclients that want to use multicast connect to an RTgms process instead of an RTserver process. The RTgms process manages the multicast message routing and connects to the RTserver. See Creating a Connection to RTgms on page 165 for more information. Note that you use a special multicast logical connection name to connect to RTgms. This is described in Logical Connection Names for Multicast on page 165.

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 for its global connection instead of connecting to an RTserver. In most cases, the only change required is to the ss.group_names and ss.server_names options for the RTclient. The RTclient still uses the TipcSvc.getSrv method to make the connection. For information on RTclient options and how to set them, see Chapter 9, RTclient Options, on page 135.

The ss.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 ss.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, the property database for your RTclient might contain:

```
ss.group_names=rtworks
ss.server_names=tcp:nodea
```

Let us 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:

```
ss.group_names=mcast1
ss.server_names=pgm:nodea
```

If you want to connect to an RTgms that is not using the default port, change the ss.server_names line to:

```
ss.server_names=pgm:nodea:tcp.6000
```

which connects to the RTgms on `nodea` using the TCP protocol on port 6000. For more information on the format of RTgms addresses, see Logical Connection Names for Multicast.

Use TipcSvc.getSrv to connect to the RTgms process the same way you connect to an RTserver process:

```
TipcSrv srv=TipcSvc.getSrv();
if (!srv.create()) {
  Tut.exitFailure("Couldn't connect to RTgms!\n");
}
```

To also connect to RTservers, the RTclient can use the multiple connections feature, and create those RTserver connections using the TipcSvc.createSrv method.

Logical Connection Names for Multicast

There are two parts of the logical connection name that differ for multicast. Generally, the logical connection name is:

protocol:*node*:*address*

When specifying a multicast logical connection name to connect to RTgms, the value for protocol is always `pgm`. The address portion of your logical connection name is a different format than for other protocols.

The format for multicast is:

`pgm`:*node*:*unicast_protocol*.*address*

where:

node — is the name of the node where RTgms is running. You can use `_node` rather than specifying a name.

unicast_protocol — specifies the unicast protocol to use when sending data to an RTgms. The valid values you can specify are `tcp` or `local`.

On Windows, the default protocol is `tcp`. On UNIX, the default protocol is `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`, so if you do not specify this field, the default provides the correct value.

For example, on UNIX:

```
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.

Chapter 11 Guaranteed Message Delivery

This chapter gives a broad overview of the SmartSockets feature of guaranteed message delivery (GMD), and then focuses on its use with Java. For an in-depth discussion of GMD and its features, see the *TIBCO SmartSockets User's Guide*.

Topics

Overview of GMD

Under normal operation in SmartSockets, all messages sent through connections are delivered successfully and processed in a timely manner. If a network failure occurs, however, data can be lost. For some applications, such as bank transactions or Internet commerce, missed messages or duplicate messages are unacceptable. With GMD, an RTclient can stay informed as to whether a message was delivered to some or all subscribers. GMD fully recovers from failures and ensures that messages are transmitted as required.

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 Features

GMD has these features:

- persistence of messages, stored safely 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 recovery procedures that you design

- easy to use

  Simply set the delivery mode property of a message using the setDeliveryMode method on the TipcMsg class or the TipcMt class, and set the delivery timeout property of a message with setDeliveryTimeout on the TipcMsg class or the TipcMt class.

- easy to configure

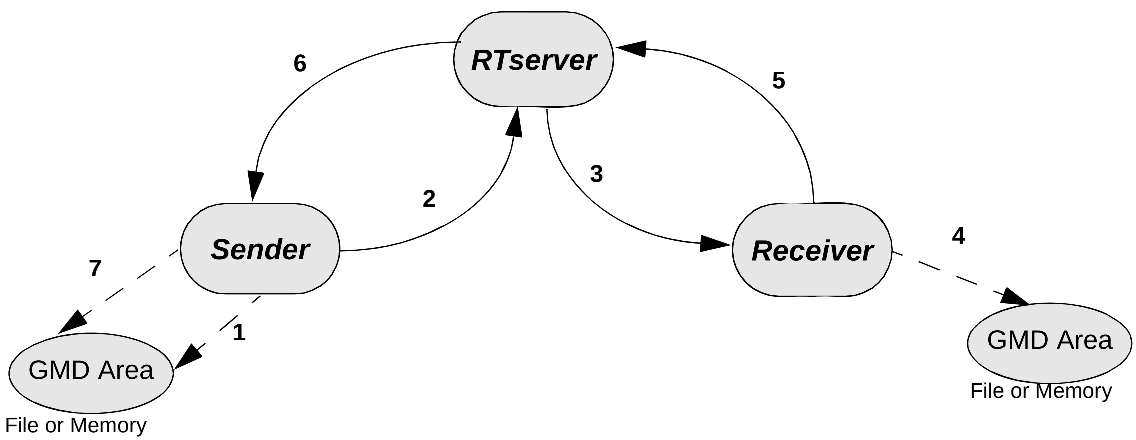
  Use the default settings, or set the few GMD-related options in your option database.

- industry benchmark performance that is limited only by the speed of your local file system and the network

How GMD Works

After you initially configure GMD for your system, certain steps occur automatically, without any intervention. Figure 10 shows the order of events:

Figure 10 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.
5. Acknowledgment message is sent to RTserver.
6. Acknowledgment message is sent to Sender.
7. Message is deleted from GMD area.

Configuring GMD

To use the default configuration for GMD, you simply start using GMD, and the default values of the GMD-related options are used. Information on starting GMD is in Sending GMD Messages on page 176. When you start GMD using the default configuration, file-based GMD is activated if you had set a value for ss.unique_subject. If there was no value set for ss.unique_subject, memory-based GMD is activated.

If you want to configure GMD to use values other than the defaults, you can set the Java GMD-related options in the options database using the same techniques as for any other options. See Option (Property) Databases and Loading RTclient Options on page 136 for information on setting options.

Java GMD-Related Options

The Java options that affect GMD, listed in Table 4, are similar to the RTclient options for C described in the *TIBCO SmartSockets User's Guide*. You might find it helpful to read the descriptions in that book for setting GMD-related options. Complete descriptions of the Java RTclient options are in Chapter 9, RTclient Options.

Table 4 Options Related to GMD

Option Name	Description
ss.ipc_gmd_directory	Specifies where the GMD files should be located.
ss.ipc_gmd_type	Specifies whether GMD is file-based or memory-based, which also determines where to create the GMD area (on disk or in memory)
ss.server_delivery_timeout	Controls the delivery timeout property of the connection to RTserver.
ss.server_disconnect_mode	Specifies the action RTserver should take when RTclient disconnects from RTserver.
ss.unique_subject	Specifying a unique_subject triggers file-based GMD.

Configuring File-Based GMD

To configure file-based GMD, you need to set some of the GMD-related options. For information on setting options, see Setting RTclient Options on page 137. Follow these steps for file-based GMD:

Step 1 **Set a value for ss.unique_subject**

You must set a value for the ss.unique_subject option, a value other than the default of *_node_start-time*. For more information, see ss.unique_subject on page 159.

Step 2 **Use the default value for ss.ipc_gmd_type**

If the ss.ipc_gmd_type option is left at its default value of `default`, file-based GMD is attempted. For more information, see ss.ipc_gmd_directory on page 144.

Step 3 **Optionally, set values for the other GMD-related options**

You can use the default settings for the ss.ipc_gmd_directory, ss.server_delivery_timeout, and ss.server_disconnect_mode options or set your own values. These options do not affect what type of GMD is attempted, file-based or memory-based. You can configure these options as you like for your system:

1. Set a value for the ss.ipc_gmd_directory option or use the default value.

   Under file-based GMD, the default location for GMD files is the directory where Java is installed. You can set the ss.ipc_gmd_directory option to specify a different directory.

2. Set a value for the ss.server_delivery_timeout option or use the default value of 30 seconds.

   You can change the value to give the sending process more or less time to wait for all receiving processes to acknowledge delivery of a guaranteed message. If you set the value to `0.0`, this option is disabled, and the sending process never times out.

3. Set a value for the ss.server_disconnect_mode option or use the default value of `gmd_failure`.

   The ss.server_disconnect_mode option specifies what the RTserver should do, if anything, if and when an RTclient disconnects from RTserver. The three possible values are:

`warm`	specifies that RTserver saves subject, project, and guaranteed message delivery information about the disconnecting RTclient so that no messages are lost.
`gmd_failure`	specifies that RTserver destroys all information about the disconnecting RTclient and causes pending guaranteed message delivery to fail.
`gmd_success`	specifies that RTserver destroys all information about the disconnecting RTclient, but causes pending guaranteed message delivery to succeed.

Notes on File-Based GMD

- File-based GMD is only attempted if both are true:
  - — the ss.unique_subject option is set to a value other than its default, and
  - — the ss.ipc_gmd_type option is set to `default`

  If ss.ipc_gmd_type is set to `memory`, memory-based GMD occurs even if you set a value for ss.unique_subject. However, if you did not set a value for ss.unique_subject, memory-based GMD is used even if ss.ipc_gmd_type is set to `default`.
- When ss.unique_subject is set, and ss.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 174.
- Although you specify file-based GMD, memory-based GMD is used whenever file-based GMD is attempted but is unsuccessful. This provides a measure of safety, because even though file-based GMD might fail, your messages are still protected under GMD.

Configuring Memory-Based GMD

To configure memory-based GMD, you need to set some of the GMD-related options. For information on setting options, see Setting RTclient Options on page 137. Follow these steps for memory-based GMD:

Step 1 **Set the value for ss.ipc_gmd_type to memory**

If the value is set to `memory`, memory-based GMD is used, even if ss.unique_subject is set. If left to its default value of `default`, file-based GMD is attempted. For more information, see ss.ipc_gmd_type on page 144.

Step 2 **Optionally, set values for the other GMD-related options**

You can use the default settings for the options ss.unique_subject, ss.server_delivery_timeout, and ss.server_disconnect_mode, or set your own values:

1. Set a value for the ss.unique_subject option or use the default value.

   If you need to specify a value for ss.unique_subject other than the default value, you can set ss.unique_subject and still get memory-based GMD if you set ss.ipc_gmd_type to `memory`.

2. Set a value for the ss.server_delivery_timeout option or use the default value of 30 seconds.

   You can change the value to give the sending process more or less time to wait for all receiving processes to acknowledge delivery of a guaranteed message. If you set the value to `0.0`, this option is disabled, and the sending process never times out.

3. Set a value for the ss.server_disconnect_mode option or use the default value of `gmd_failure`.

   The ss.server_disconnect_mode option specifies what the RTserver should do, if anything, if and when an RTclient disconnects from RTserver. The three possible values are:

   `warm` specifies that RTserver saves subject, project, and guaranteed message delivery information about the disconnecting RTclient so that no messages are lost.

   `gmd_failure` specifies that RTserver destroys all information about the disconnecting RTclient and causes pending guaranteed message delivery to fail.

   `gmd_success` specifies that RTserver destroys all information about the disconnecting RTclient, but causes pending guaranteed message delivery to succeed.

Reverting to Memory-Based GMD

When ss.unique_subject is set, and ss.ipc_gmd_type is set to `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. If the disk files cannot be created or a security exception is thrown, Java displays a warning message to the console, and automatically switches to memory-based GMD.

A common example of this is a Java applet. Most Java applets cannot write to the local file system, where the GMD spool area resides for file-based GMD. If you specify file-based GMD, the applet attempts to write to the disk where the GMD area is located. A SecurityException is thrown, and SmartSockets reverts to memory-based GMD.

Using GMD

File-based GMD and memory-based GMD operate in exactly the same way. Whenever a GMD message is sent, a copy is stored in the GMD spool area, on disk or in memory, based on the values you specified for the options during configuration. The files for GMD are created automatically and only when necessary, typically on the first publish or first reception of a GMD message.

Java GMD Methods

The Java classes and methods used with GMD are listed in Table 5. Full API reference information for these classes and methods is online in Javadoc format and is provided with the SmartSockets product.

Table 5 Java Classes and Methods for GMD

Name	Description
TipcMsg.getDeliveryMode	Get the delivery mode of a message. Equivalent C function: TipcMsgGetDeliveryMode
TipcMt.getDeliveryMode	Get the delivery mode of a message type. Equivalent C function: TipcMtGetDeliveryMode

Table 5 Java Classes and Methods for GMD

Name	Description
TipcMsg.getDeliveryTimeout	Get the delivery timeout of a message in seconds. Equivalent C function: TipcMsgGetDeliveryTimeout
TipcMt.getDeliveryTimeout	Get the delivery timeout of a message type in seconds. Equivalent C function: TipcMtGetDeliveryTimeout
TipcMsg.setDeliveryMode	Set the delivery mode of a message. Overrides the value set for a message by TipcMt.setDeliveryMode. GMD settings: DELIVERY_SOME or DELIVERY_ALL Equivalent C function: TipcMsgSetDeliveryMode
TipcMt.setDeliveryMode	Set the delivery mode of a message type. GMD settings: DELIVERY_SOME or DELIVERY_ALL Equivalent C function: TipcMtSetDeliveryMode
TipcMsg.setDeliveryTimeout	Set the delivery timeout of a message in seconds. Overrides the value set for a message by TipcMt.setDeliveryTimeout. GMD settings: `0.0` disables checking for delivery timeouts. The value can be any real number 0.0 or greater. Equivalent C function: TipcMsgSetDeliveryTimeout
TipcMt.setDeliveryTimeout	Set the delivery timeout of a message type. GMD settings: `0.0` disables checking for delivery timeouts. The value can be any real number 0.0 or greater. Equivalent C function: TipcMtSetDeliveryTimeout
TipcSrv.gmdFileDelete	Delete guaranteed message delivery files for the connection to RTserver. Useful for deleting any obsolete GMD information. Equivalent C function: TipcSrvGmdFileDelete
TipcSrv.gmdMsgAck	Acknowledge the delivery of a message. Equivalent C function: TipcMsgAck
TipcSrv.getGmdNumPending	Get the number of outgoing GMD messages still pending on a connection. Equivalent C function: TipcSrvGetGmdNumPending

Sending GMD Messages

To send messages with GMD, simply set the message delivery mode to TipcDefs.DELIVERY_SOME or to TipcDefs.DELIVERY_ALL. Send the message as usual with TipcSrv.send.

There are two ways to set the message delivery mode:

- use TipcMsg.setDeliveryMode to set the mode for individual messages
- use TipcMt.setDeliveryMode to set the mode for all messages of a particular message type

Setting the mode to TipcDefs.DELIVERY_SOME means that the guaranteed message must go to at least one subscriber. That is, the sending process only needs to receive an acknowledgment from one receiving process before timing out in order to declare success. It can receive acknowledgments from more than one receiving process.

Setting the mode to TipcDefs.DELIVERY_ALL means that the guaranteed message must go to all the subscribers. That is, the sending process must receive acknowledgments from all the receiving processes before timing out before it can declare success.

For example:

```
msg.setDeliveryMode(TipcDefs.DELIVERY_ALL);
srv.send(msg);
```

For GMD, TipcSrv.send:

1. Increments an internal per-connection outgoing sequence number.
2. Sets the message sequence number to the incremented value.
3. Saves a copy of the message in the connection GMD area.
4. Saves the current wall clock time in the GMD area, for detecting a delivery timeout.

Receiving GMD Messages

For GMD, TipcSrv.next recognizes a message resent with GMD 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 might overflow and wrap around the four byte sequence number. If the resent message has already been acknowledged, TipcSrv.next acknowledges the message again so that the sender is notified this time of successful delivery.

TipcSrv.next always allows GMD messages that have not been resent to pass through, regardless of their sequence number. This allows flexibility and correct behavior when some processes use TipcSrv.gmdFileDelete and others do not, enabling use of old sequence numbers.

TipcSrv.next 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.

Acknowledging GMD Messages

Typically, an application calls TipcSvc.mainLoop and this acknowledges the message (sends a GMD_ACK message).

Also, you can useTipcMsg.ack, which automatically calls TipcSrv.gmdMsgAck to acknowledge the message for GMD. TipcSrv.gmdMsgAck can also be called manually to acknowledge a message. For example:

```
srv.gmdMsgAck(msg)
```

TipcSrv.gmdMsgAck 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.

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 TipcSrv.mainLoop or TipcSrv.next before the program exits. A sending process can also check how many outgoing GMD messages are still pending with TipcSrv.getGmdNumPending. This is useful for waiting until all acknowledgments arrive. For example:

```
System.out.println("Read data until all acknowledgments come in.");
do {
  srv.mainLoop(1.0);
  num_pending=srv.getGmdNumPending();
} while (num_pending > 0);
```

Example of Using GMD

Here is an example of configuring and using GMD with Java. This example also uses a callback to process GMD failures. See Processing of GMD_FAILURE Messages on page 182 to see the sample code for the callback.

```
import java.io.*;
import com.smartsockets.*;

public class gmd_example {

  private static final int SAMPLE = 1001;

/*================================================================ */
  public gmd_example() {

    TipcSrv srv=TipcSvc.getSrv();

  // set the server names
    try {
      Tut.setOption("ss.server_names", "altoids,maple");
    }
    catch (TipcException e) {
      Tut.fatal(e);
    } // catch

    // set the unique subject for gmd
    try {
      Tut.setOption("ss.unique_subject", "gmd_publisher");
    }
    catch (TipcException e) {
      Tut.fatal(e);
    } // catch

   // delete old gmd files
    try {
      srv.gmdFileDelete();
    }
    catch (TipcException e) {
      Tut.fatal(e);
    } // catch

   // connect to RTserver
    try {
      srv.create();
    }
    catch (TipcException e) {
      Tut.exitFailure("Couldn't connect to RTserver!");
    } // catch

  // get message type for gmd failure message
    TipcMt mt = TipcSvc.lookupMt(TipcMt.GMD_FAILURE);
```

```
 //destroy old gmd callback
  TipcCb cb = srv.getDefaultGmdFailureCb();
  try {
    srv.removeProcessCb(cb);
  }
  catch (TipcException e) {
    Tut.fatal(e);
  } // catch

 // setup new callback--callback code is shown later in this chapter
  gmdFailureMsgCallback pcb = new gmdFailureMsgCallback();
  TipcCb pcbh = srv.addProcessCb(pcb, mt, srv);
// check the 'handle' returned for validity
  if (null == pcbh) {
    Tut.exitFailure("Couldn't register gmd failure callback!");
  } // if

// define new message type
  try {
    mt = TipcSvc.createMt("SAMPLE", SAMPLE, "str");
  }
  catch (TipcException e) {
    Tut.exitFailure("Message type already exists!");
  } // catch

// following is a for loop which publishes 3 messages.
// assuming that no one subscribes to /sample/gmd and
// the option in the RTserver you connect to is set as follows:
// setopt zero_recv_failure_option TRUE
// each message will result in an immediate gmd failure
// and the callback entered for each

  for (int i=1; i <= 3; i++) {

    try {
     // create a message of type SAMPLE
      TipcMsg msg = TipcSvc.createMsg(mt);
      msg.setDest("/sample/gmd");                     // publish to subject
      msg.setDeliveryMode(TipcDefs.DELIVERY_ALL);   // all receivers to ack
      msg.setDeliveryTimeout(0.1);
      msg.addNamedStr("Data", "gmd message #" + i);
      System.out.println("< sending gmd message #" + i + ">");

     // send and flush the message
      srv.send(msg);
      srv.flush();

 // call mainloop to read in acknowledgement message
      srv.mainLoop(2.0);
    }
    catch (TipcException e) {
      Tut.fatal(e);
    } // catch

  } // for
```

```
    //disconnect from the server
    try {
      srv.destroy();
      srv.removeProcessCb(pcbh);    //unregister the callback
    }
    catch (TipcException e) {
      Tut.exitFailure("unable to disconnect from server");
    } // catch

  } // gmd_example (constructor)

  public static void main(String[] argv) {
    new gmd_example();
  } // main
} // gmd_example class
```

Handling GMD Failures

Recovery from GMD_FAILURE messages is highly specific to the application, 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 needed. SmartSockets by default outputs a warning, terminates GMD for the failed message, and continues.

GMD_FAILURE Messages

When GMD fails, a GMD_FAILURE message is created internally by SmartSockets. TipcSrv.process is called to process the message and 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, terminating 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, which is 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

The only type of GMD_FAILURE message produced for non-RTclient or non-RTserver GMD is a delivery timeout failure. The third field of the GMD_FAILURE message is TipcSrv.ERROR_GMD_SENDER_TIMEOUT.

Connections automatically check for delivery timeouts whenever data is read from the connection (with TipcSrv.next) or the connection is checked to see if data can be read (with TipcSrv.check). You must use TipcSrv.next and TipcSrv.check frequently enough.

Processing of GMD_FAILURE Messages

The default GMD failure callback is a sample callback designed only to warn the user that guaranteed delivery of a message has failed. You should create applications that destroy the callback and create their own process callbacks for GMD_FAILURE messages, performing actions such as a recovery procedures you design.

To obtain the default callback for GMD_FAILURE, use the TipcSrv.getDefaultGmdFailureCb method. To register a new GMD_FAILURE callback, use the TipcSrv.addProcessCb with a message type of TipcMt.GMD_FAILURE.

Here is an example of a process callback:

```
import java.io.*;
import com.smartsockets.*;

public class gmd_example {

  private static final int SAMPLE = 1001;

/*=================================================================== */
/*..gmdFailureMsgCallback - callback for gmd failure                 */
  public class gmdFailureMsgCallback implements TipcProcessCb {

    public void process(TipcMsg msg, Object arg) {

      System.out.println("GMD failure");
      TipcMsg sender_msg = null;

     //* get published message header and contents */
      try {
        sender_msg = msg.nextMsg();
      }
      catch (TipcException e) {
        Tut.fatal(e);
      } // catch

    // point to error id field
      try {
        msg.setCurrent(2);
      }
      catch (TipcException e) {
        Tut.fatal(e);
      } // catch

      int error_number = 0;

      try {
        error_number=msg.nextInt4();
      }
      catch (TipcException e) {
        Tut.fatal(e);
      } // catch
```

```
      // print out message saying what happened
      switch (error_number) {
        case 518:
          System.out.println("GMD sender timed out");
          break;
        case 519:
          System.out.println("GMD receiver timed out");
          break;
        case 520:
          System.out.println("GMD receiver exited");
          break;
        case 521:
          System.out.println("No receivers for subject: "
                             + sender_msg.getDest() + "");
          break;
        default:
          System.out.println("Unknown error code: "
                             + error_number);
          break;
      }

      // remove copies of message
      try {
        TipcSrv srv=TipcSvc.getSrv();
        srv.gmdMsgServerDelete(sender_msg);  // delete from server
        srv.gmdMsgDelete(sender_msg);        // delete from local spool
      }
      catch (TipcException e) {
        Tut.fatal(e);
      } // catch

      System.out.println("FAILED MESSAGE Follows");
      sender_msg.print();

    } // process callback
  } // gmdFailureMsgCallback
```

File-Based GMD Considerations

If you plan to use file-based GMD, you can control the resending of spooled GMD messages. You can force all spooled GMD messages to be resent, or you can prevent them from being resent by deleting them from the spool.

Resending GMD Messages

When a Java RTclient comes up and a connection is made to RTserver, the Java SmartSockets class library automatically resends any GMD messages that are stored on disk. This automatic sending of messages ensures that if a Java RTclient crashed and then came back up, any messages that were not acknowledged by a GMD_ACK, and are still in the GMD spool area, are automatically resent.

To force all spooled GMD messages to be resent, invoke the method TipcSrv.gmdResend. This method must be invoked after the connection to RTserver is established. For RTserver connections, this is done automatically once the connection to RTserver is established.

Removing GMD Files

To prevent the SmartSockets Java RTclient from resending spooled GMD messages when connecting to RTserver, invoke the method gmdFileDelete from RTclient. You must invoke gmdFileDelete before the connection to RTserver is established, but after the unique subject has been set. For example:

```
public class test {
    public static void main(String[] args) {
    TipcSrv srv;

    try {
      // load options file, which must have value for
      // ss.unique_subject
      Tut.loadOptionsFile("test.cm");

      // get server instance
      srv = TipcSvc.getSrv();

      // remove GMD files from spool area
      srv.gmdFileDelete();

      // connect to server, which would normally resend any
      // unacknowledged messages from the GMD spool area, if
      // present
      srv.create();

      // continue processing...
    }
    catch (Exception e) {
      System.out.println("Caught exception " + e);
      System.out.println(e.printStackTrace());
      return;
    }
  }
}
```

Warm Connections

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 (that is, 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 functions behave in a fashion similar to when a full connection exists.

There are two types of warm connections:

- a connection that is created as a warm connection, when RTclient connects to an RTserver using:

  ```
  srv.create(TipcSrv.CONN_WARM);
  ```

- a connection that starts out as a full connection and changes to a warm connection when the connection to RTserver is disconnected or destroyed. This is frequently referred to as a connection with a warm RTclient, because the RTserver remembers information about the RTclient.

New Warm Connections

A new warm connection is created when the RTclient creates a warm connection to RTserver. The RTserver is not aware of the RTclient when the RTclient has a warm connection. With a warm connection to RTserver, callbacks can be created, callbacks can be destroyed, and messages can be buffered, including messages sent without GMD. If RTclient has a warm connection and then creates a full connection (the connection changes from warm to full), the warm-buffered messages are flushed to the newly-created full connection. For more information, see the section on warm connections to RTserver in the *TIBCO SmartSockets User's Guide*.

Creating a New Warm Connection

To create a warm connection to RTserver, use TipcSvc.create with the argument set to CONN_WARM. Here is a simple example that sets the ss.server_auto_connect option to `FALSE`, creates a warm connection, subscribes to subjects, and then creates a full connection. The RTclient subscribes to subjects `/subj0` to `/subj99`, but only establishes the full connection after it is done subscribing:

```
/*------------------------java example-----------------------------

import com.smartsockets.*;

public class test {
  test() {
    TipcSrv srv = null;
    int i;
    try {
      /*
       * set the server_auto_connect option to false so
       * client does not automatically try and create
       * a connection when it calls any of the srv methods.
       */
      Tut.setOption("ss.server_auto_connect", "false");

      /*
       * get a srv object
       */
      srv = TipcSvc.getSrv();

      /*
       * create a warm connection
       */
      srv.create(TipcSrv.CONN_WARM);

      /*
       * subscribe to all the subjects
       */
      for (i=0; i<100; i++) {
        String sub = "/subj" + i;
        System.out.println("subscribing to = " + sub);
        srv.setSubjectSubscribe(sub, true);
      }
```

```
    /*
     * create a full connection
     */
        srv.create();
        }
        catch (TipcException e) {
          e.printStackTrace();
        }
    }

    public static void main(String args[]) {
      test client = new test();
    }
}
```

You can use TipcSvc.destroy to destroy warm connections. For more information on creating and destroying warm connections, look up TipcSvc in the Java API reference, provided online in Javadoc format.

Connections with Warm RTclients

A warm connection with a warm RTclient starts out as a full connection, and becomes a warm connection when the full connection is lost or destroyed. This type of warm connection is used only for GMD. With a warm RTclient, the RTserver remembers the name of the RTclient and the subjects to which the RTclient was subscribing when that RTclient disconnected. RTserver tracks the GMD messages that this warm RTclient should receive and acknowledge. RTserver does not buffer any non-GMD messages for the RTclient.

RTclient informs RTserver to keep warm RTclient information for itself by setting the ss.server_disconnect_mode option to `warm` before creating or destroying a full connection (the value of ss.server_disconnect_mode is sent to RTserver at those times). In this warm mode, if an RTclient disconnects for any reason (crashes or simply calls TipcSrvDestroy), all necessary RTservers (those with direct GMD publishing RTclients) keep warm RTclient information.

The warm RTclient is not 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.

For more information on warm RTclients, see the section on warm RTclient in RTserver in the *TIBCO SmartSockets User's Guide*.

Creating a Warm RTclient

You do not create a warm connection with a warm RTclient. Instead, you configure a warm RTclient by setting the value for the ss.server_disconnect_mode option to `warm`. Then, when the full connection from that RTclient to an RTserver is disconnected, it changes to a warm connection automatically. For more information on setting the option, see Configuring GMD on page 170.

GMD Limitations

GMD can recover from most network failures, but in particular cases, there can be problems:

- 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 might 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-sequential order, the resent messages might be skipped.

Though unlikely, these conditions can occur if a node or disk crashes while a message is being written or while sequence numbers are being updated.

Appendix A Java API to C API Mapping

The SmartSockets Java Class Library includes Java wrappers for many SmartSockets C functions. This appendix shows how these Java methods map to the original C function. This can help you look up the C function for more information on the API in general. See the *TIBCO SmartSockets Application Programming Interface* for complete documentation of all C functions.

Topics

Table 6   Interface TipcConnClient

Java Method Name	C Function Name	Comments and Exceptions
addDefaultCb	TipcConnDefaultCbCreate	
addErrorCb	TipcConnErrorCbCreate	
addProcessCb	TipcConnProcessCbCreate	
addQueueCb	TipcConnQueueCbCreate	
addReadCb	TipcConnReadCbCreate	
addWriteCb	TipcConnWriteCbCreate	
check	TipcConnCheck	
destroy	TipcConnDestroy	
flush	TipcConnFlush	
getArch	TipcConnGetArch	
getAutoFlushSize	TipcConnGetAutoFlushSize	
getBlockMode	TipcConnGetBlockMode	
getDefaultControlCb		No parallel C function exists.
getDefaultGmdFailureCb		No parallel C function exists.
getGmdDir	TipcGetGmdDir	
getGmdNumPending	TipcConnGetGmdNumPending	
getNode	TipcConnGetNode	
getNumQueued	TipcConnGetNumQueued	
getOption		No parallel C function exists.
getOptionBool		No parallel C function exists.
getOptionDouble		No parallel C function exists.
getOptionInt		No parallel C function exists.

Java Method Name	C Function Name	Comments and Exceptions
getOptionStr		No parallel C function exists.
getPid	TipcConnGetPid	
getProperties		No parallel C function exists.
getQueueSize		No parallel C function exists.
getReadBufferSize	TipcConnBufferGetReadSize	
getTimeout	TipcConnGetTimeout	
getTrafficBytesRecv	TipcConnTrafficGetBytesRecv	
getTrafficBytesSent	TipcConnTrafficGetBytesSent	
getTrafficMsgsRecv	TipcConnTrafficGetMsgsRecv	
getTrafficMsgsSent	TipcConnTrafficGetMsgsSent	
getUniqueSubject	TipcConnGetUniqueSubject	
getUser	TipcConnGetUser	
getWriteBufferSize	TipcConnBufferGetWriteSize	
gmdFileDelete	TipcConnGmdFileDelete	
gmdMsgAck		No parallel C function exists.
gmdMsgDelete	TipcConnGmdMsgDelete	
gmdMsgResend	TipcConnGmdMsgResend	
gmdResend	TipcConnGmdResend	
insert	TipcConnMsgInsert	
keepAlive	TipcConnKeepAlive	
loadOptionsFile		No parallel C function exists.
loadOptionsStream		No parallel C function exists.
loadOptionsURL		No parallel C function exists.

Java Method Name	C Function Name	Comments and Exceptions
lookupDefaultCb	TipcConnDefaultCbLookup	
lookupErrorCb	TipcConnErrorCbLookup	
lookupProcessCb	TipcConnProcessCbLookup	
lookupQueueCb	TipcConnQueueCbLookup	
lookupReadCb	TipcConnReadCbLookup	
lookupWriteCb	TipcConnWriteCbLookup	
mainLoop	TipcConnMainLoop	
makeSubjectAbsolute		No parallel C function exists.
next	TipcConnMsgNext	
process	TipcConnMsgProcess	
read	TipcConnRead	
removeDefaultCb		No parallel C function exists.
removeErrorCb		No parallel C function exists.
removeProcessCb		No parallel C function exists.
removeQueueCb		No parallel C function exists.
removeReadCb		No parallel C function exists.
removeWriteCb		No parallel C function exists.
search	TipcConnMsgSearch	
searchType	TipcConnMsgSearchType	
send	TipcConnMsgSend	
sendRpc	TipcConnMsgSendRpc	
setAutoFlushSize	TipcConnSetAutoFlushSize	
setOption		No parallel C function exists.

Java Method Name	C Function Name	Comments and Exceptions
setTimeout	TipcConnSetTimeout	

Table 7   Interface TipcConnServer

Java Method Name	C Function Name	Comments and Exceptions
accept	TipcConnAccept	
destroy		No parallel C function exists.

Table 8   Class TipcMon

Java Method Name	C Function Name
clientBufferPoll	TipcMonClientBufferPoll
clientCbPoll	TipcMonClientCbPoll
clientCpuPoll	TipcMonClientCpuPoll
clientExtPoll	TipcMonClientExtPoll
clientGeneralPoll	TipcMonClientGeneralPoll
clientInfoPoll	TipcMonClientInfoPoll
clientMsgTrafficPoll	TipcMonClientMsgTrafficPoll
clientMsgTypePoll	TipcMonClientMsgTypePoll
clientNamesNumPoll	TipcMonClientNamesNumPoll
clientNamesPoll	TipcMonClientNamesPoll
clientOptionPoll	TipcMonClientOptionPoll
clientSubjectPoll	TipcMonClientSubjectPoll
clientSubscribeNumPoll	TipcMonClientSubscribeNumPoll
clientSubscribePoll	TipcMonClientSubscribePoll
clientTimePoll	TipcMonClientTimePoll

Java Method Name	C Function Name
getClientCongestionWatch	TipcMonClientCongestionGetWatch
getClientBufferWatch	TipcMonClientBufferGetWatch
getClientMsgRecvWatch	TipcMonClientMsgRecvGetWatch
getClientMsgSendWatch	TipcMonClientMsgSendGetWatch
getClientNamesWatch	TipcMonClientNamesGetWatch
getClientSubscribeWatch	TipcMonClientSubscribeGetWatch
getClientTimeWatch	TipcMonClientTimeGetWatch
getIdentStr	TipcMonGetIdentStr
getProjectNamesWatch	TipcMonProjectNamesGetWatch
getServerCongestionWatch	TipcMonServerCongestionGetWatch
getServerConnWatch	TipcMonServerConnGetWatch
getServerMaxClientLicensesWatch	TipcMonServerMaxClientLicensesGetWatch
getServerNamesWatch	TipcMonServerNamesGetWatch
getSubjectNamesWatch	TipcMonSubjectNamesGetWatch
getSubjectSubscribeWatch	TipcMonSubjectSubscribeGetWatch
projectNamesPoll	TipcMonProjectNamesPoll
serverBufferPoll	TipcMonServerBufferPoll
serverConnPoll	TipcMonServerConnPoll
serverCpuPoll	TipcMonServerCpuPoll
serverGeneralPoll	TipcMonServerGeneralPoll
serverMsgTrafficPoll	TipcMonServerMsgTrafficPoll
serverNamesPoll	TipcMonServerNamesPoll
serverOptionPoll	TipcMonServerOptionPoll

Java Method Name	C Function Name
serverRoutePoll	TipcMonServerRoutePoll
serverTimePoll	TipcMonServerTimePoll
setClientBufferWatch	TipcMonClientBufferSetWatch
setClientCongestionWatch	TipcMonClientCongestionSetWatch
setClientMsgRecvWatch	TipcMonClientMsgRecvSetWatch
setClientMsgSendWatcg	TipcMonClientMsgSendSetWatch
setClientNamesWatch	TipcMonClientNamesSetWatch
setClientSubscribeWatch	TipcMonClientSubscribeSetWatch
setClientTimeWatch	TipcMonClientTimeSetWatch
setIdentStr	TipcMonSetIdentStr
setProjectNamesWatch	TipcMonProjectNamesSetWatch
setServerCongestionWatch	TipcMonServerCongestionSetWatch
setServerConnWatch	TipcMonServerConnSetWatch
setServerMaxClientLicensesWatch	TipcMonServerMaxClientLicensesSetWatch
setServerNamesWatch	TipcMonServerNamesSetWatch
setSubjectNamesWatch	TipcMonSubjectNamesSetWatch
setSubjectSubscribeWatch	TipcMonSubjectSubscribeSetWatch
subjectNamesPoll	TipcMonSubjectNamesPoll
subjectSubscribePoll	TipcMonSubjectSubscribePoll

Table 9 Class TipcMonExt

Java Method Name	C Function Name
delete	TipcMonExtDelete
setBinary	TipcMonExtSetBinary
setBool	TipcMonExtSetBool
setBoolArray	TipcMonExtSetBoolArray
setInt2	TipcMonExtSetInt2
setInt2Array	TipcMonExtSetInt2Array
setInt4	TipcMonExtSetInt4
setInt4Array	TipcMonExtSetInt4Array
setInt8	TipcMonExtSetInt8
setInt8Array	TipcMonExtSetInt8Array
setReal4	TipcMonExtSetReal4
setReal4Array	TipcMonExtSetReal4Array
setReal8	TipcMonExtSetReal8
setReal8Array	TipcMonExtSetReal8Array
setStr	TipcMonExtSetStr
setStrArray	TipcMonExtSetStrArray
setUtf8	TipcMonExtSetUtf8
setUtf8Array	TipcMonExtSetUtf8Array

Table 10 Interface TipcMsg

Java Method Name	C Function Name	Comments and Exceptions
ack	TipcMsgAck	
addNamedBinary	TipcMsgAddNamedBinary	
addNamedBool	TipcMsgAddNamedBool	
addNamedBoolArray	TipcMsgAddNamedBoolArray	
addNamedByte	TipcMsgAddNamedByte	
addNamedChar	TipcMsgAddNamedChar	
addNamedGuid		No parallel C function exists.
addNamedGuidArray		No parallel C function exists.
addNamedInt2	TipcMsgAddNamedInt2	
addNamedInt2Array	TipcMsgAddNamedInt2Array	
addNamedInt4	TipcMsgAddNamedInt4	
addNamedInt4Array	TipcMsgAddNamedInt4Array	
addNamedInt8	TipcMsgAddNamedInt8	
addNamedInt8Array	TipcMsgAddNamedInt8Array	
addNamedMsg	TipcMsgAddNamedMsg	
addNamedMsgArray	TipcMsgAddNamedMsgArray	
addNamedMsgId		No parallel C function exists.
addNamedMsgIdArray		No parallel C function exists.
addNamedObject		No parallel C function exists.
addNamedReal4	TipcMsgAddNamedReal4	
addNamedReal4Array	TipcMsgAddNamedReal4Array	
addNamedReal8	TipcMsgAddNamedReal8	

Java Method Name	C Function Name	Comments and Exceptions
addNamedReal8Array	TipcMsgAddNamedReal8Array	
addNamedStr	TipcMsgAddNamedStr	
addNamedStrArray	TipcMsgAddNamedStrArray	
addNamedTimestamp	TipcMsgAddNamedTimestamp	
addNamedTimestampArray	TipcMsgAddNamedTimestamp Array	
addNamedUnknown	TipcMsgAddNamedUnknown	
addNamedUtf8	TipcMsgAddNamedUtf8	
addNamedUtf8Array	TipcMsgAddNamedUtf8Array	
addNamedXml	TipcMsgAddNamedXml	
appendBinary	TipcMsgAppendBinary	
appendBool	TipcMsgAppendBool	
appendBoolArray	TipcMsgAppendBoolArray	
appendByte	TipcMsgAppendByte	
appendChar	TipcMsgAppendChar	
appendGuid		No parallel C function exists.
appendGuidArray		No parallel C function exists.
appendInt2	TipcMsgAppendInt2	
appendInt2Array	TipcMsgAppendInt2Array	
appendInt4	TipcMsgAppendInt4	
appendInt4Array	TipcMsgAppendInt4Array	
appendInt8	TipcMsgAppendInt8	
appendInt8Array	TipcMsgAppendInt8Array	
appendMsg	TipcMsgAppendMsg	

Java Method Name	C Function Name	Comments and Exceptions
appendMsgArray	TipcMsgAppendMsgArray	
appendMsgId		No parallel C function exists.
appendMsgIdArray		No parallel C function exists.
appendObject		No parallel C function exists.
appendReal4	TipcMsgAppendReal4	
appendReal4Array	TipcMsgAppendReal4Array	
appendReal8	TipcMsgAppendInt8	
appendReal8Array	TipcMsgAppendInt8Array	
appendStr	TipcMsgAppendStr	
appendStrArray	TipcMsgAppendStrArray	
appendTimestamp	TipcMsgAppendTimestamp	
appendTimestampArray	TipcMsgAppendTimestampArray	
appendUnknown	TipcMsgAppendUnknown	
appendUtf8	TipcMsgAppendUtf8	
appendUtf8Array	TipcMsgAppendUtf8Array	
appendXml	TipcMsgAppendXml	
nextByte	TipcMsgNextByte	
clone	TipcMsgClone	
deleteCurrent	TipcMsgDeleteCurrent	
deleteField	TipcMsgDeleteField	
deleteNamedField	TipcMsgDeleteNamedField	
deleteProp		No parallel C function exists.
existsNamed	TipcMsgExistsNamed	

Java Method Name	C Function Name	Comments and Exceptions
generateMessageId	TipcMsgGenerateMessageId	
getArrivalTimestamp	TipcMsgGetArrivalTimestamp	
getCompression	TipcMsgGetCompression	
getCorrelationId	TipcMsgGetCorrelationId	
getCurrent		No parallel C function exists.
getCurrentFieldCharFormat		No parallel C function exists.
getCurrentFieldIntFormat		No parallel C function exists.
getCurrentFieldKnown	TipcMsgGetCurrentFieldKnown	
getCurrentFieldRealFormat		No parallel C function exists.
getDeliveryMode	TipcMsgGetDeliveryMode	
getDeliveryTimeout	TipcMsgGetDeliveryTimeout	
getDest	TipcMsgGetDest	
getExpiration	TipcMsgGetExpiration	
getLbMode	TipcMsgGetLbMode	
getLocalDelivery		No parallel C function exists.
getMessageId	TipcMsgGetMessageId	
getNameCurrent	TipcMsgGetNameCurrent	
getNamedBinary	TipcMsgGetNamedBinary	
getNamedBool	TipcMsgGetNamedBool	
getNamedBoolArray	TipcMsgGetNamedBoolArray	
getNamedByte	TipcMsgGetNamedByte	
getNamedChar	TipcMsgGetNamedChar	
getNamedFieldKnown		No parallel C function exists.

Java Method Name	C Function Name	Comments and Exceptions
getNamedGuid		No parallel C function exists.
getNamedGuidArray		No parallel C function exists.
getNamedInt2	TipcMsgGetNamedInt2	
getNamedInt2Array	TipcMsgGetNamedInt2Array	
getNamedInt4	TipcMsgGetNamedInt4	
getNamedInt4Array	TipcMsgGetNamedInt4Array	
getNamedInt8	TipcMsgGetNamedInt8	
getNamedInt8Array	TipcMsgGetNamedInt8Array	
getNamedMsg	TipcMsgGetNamedMsg	
getNamedMsgArray	TipcMsgGetNamedMsgArray	
getNamedMsgId		No parallel C function exists.
getNamedMsgIdArray		No parallel C function exists.
getNamedObject		No parallel C function exists.
getNamedReal4	TipcMsgGetNamedReal4	
getNamedReal4Array	TipcMsgGetNamedReal4Array	
getNamedReal8	TipcMsgGetNamedReal8	
getNamedReal8Array	TipcMsgGetNamedReal8Array	
getNamedStr	TipcMsgGetNamedStr	
getNamedStrArray	TipcMsgGetNamedStrArray	
getNamedTimestamp	TipcMsgGetNamedTimestamp	
getNamedTimestampArray	TipcMsgGetNamedTimestamp Array	
getNamedUnknown	TipcMsgGetNamedUnknown	
getNamedUtf8	TipcMsgGetNamedUtf8	

Java Method Name	C Function Name	Comments and Exceptions
getNamedUtf8Array	TipcMsgGetNamedUtf8Array	
getNamedXml	TipcMsgGetNamedXml	
getNumFields	TipcMsgGetNumFields	
getPacketSize	TipcMsgGetPacketSize	
getPriority	TipcMsgGetPriority	
getPropBinary		No parallel C function exists.
getPropInt		No parallel C function exists.
getPropShort		No parallel C function exists.
getPropStr		No parallel C function exists.
getPropStrArray		No parallel C function exists.
getReadOnly	TipcMsgGetReadOnly	
getReplyTo	TipcMsgGetReplyTo	
getSender	TipcMsgGetSender	
getSenderTimestamp	TipcMsgGetSenderTimestamp	
getSeqNum	TipcMsgGetSeqNum	
getType	TipcMsgGetType	
getTypeNamed	TipcMsgGetTypeNamed	
getUserProp	TicpMsgGetUserProp	
nextBinary	TipcMsgNextBinary	
nextBool	TipcMsgNextBool	
nextBoolArray	TipcMsgNextBoolArray	
nextChar	TipcMsgNextChar	
nextGuid		No parallel C function exists.

Java Method Name	C Function Name	Comments and Exceptions
nextGuidArray		No parallel C function exists.
nextInt2	TipcMsgNextInt2	
nextInt2Array	TipcMsgNextInt2Array	
nextInt4	TipcMsgNextInt4	
nextInt4Array	TipcMsgNextInt4Array	
nextInt8	TipcMsgNextInt8	
nextInt8Array	TipcMsgNextInt8Array	
nextMsg	TipcMsgNextMsg	
nextMsgArray	TipcMsgNextMsgArray	
nextMsgId		No parallel C function exists.
nextMsgIdArray		No parallel C function exists.
nextObject		No parallel C function exists.
nextReal4	TipcMsgNextReal4	
nextReal4Array	TipcMsgNextReal4Array	
nextReal8	TipcMsgNextReal8	
nextReal8Array	TipcMsgNextReal8Array	
nextStr	TipcMsgNextStr	
nextStrArray	TipcMsgNextStrArray	
nextTimestamp	TipcMsgNextTimestamp	
nextTimestampArray	TipcMsgNextTimestampArray	
nextType	TipcMsgNextType	
nextUnknown	TipcMsgNextUnknown	
nextUtf8	TipcMsgNextUtf8	

Java Method Name	C Function Name	Comments and Exceptions
nextUtf8Array	TipcMsgNextUtf8Array	
nextXml	TipcMsgNextXml	
print	TipcMsgPrint	
setArrivalTimestamp	TipcMsgSetArrivalTimestamp	
setCompression	TipcMsgSetCompression	
setCorrelationId	TipcMsgSetCorrelationId	
setCurrent	TipcMsgSetCurrent	
setDeliveryMode	TipcMsgSetDeliveryMode	
setDeliveryTimeout	TipcMsgSetDeliveryTimeout	
setDestTipcMsgSetDest		No parallel C function exists.
setExpiration	TipcMsgSetExpiration	
setLbMode	TipcMsgSetLbMode	
setLocalDelivery		No parallel C function exists.
setNameCurrent	TipcMsgSetNameCurrent	
setNumFields	TipcMsgSetNumFields	
setPriority	TipcMsgSetPriority	
setPropBinary		No parallel C function exists.
setPropInt		No parallel C function exists.
setPropShort		No parallel C function exists.
setPropStr		No parallel C function exists.
setPropStrArray		No parallel C function exists.
setReplyTo	TipcMsgSetReplyTo	
setSender	TipcMsgSetSender	

Java Method Name	C Function Name	Comments and Exceptions
setSenderTimestamp	TipcMsgSetSenderTimestamp	
setSeqNum		No parallel C function exists.
setType	TipcMsgSetType	
setUserProp	TipcMsgSetUserProp	
toByteArray		No parallel C function exists.
updateNamedBinary	TipcMsgUpdateNamedBinary	
updateNamedBool	TipcMsgUpdateNamedBool	
updateNamedBoolArray	TipcMsgUpdateNamedBoolArray	
updateNamedByte	TipcMsgUpdateNamedByte	
updateNamedChar	TipcMsgUpdateNamedChar	
updateNamedGuid		No parallel C function exists.
updateNamedGuidArray		No parallel C function exists.
updateNamedInt2	TipcMsgUpdateNamedInt2	
updateNamedInt2Array	TipcMsgUpdateNamedInt2Array	
updateNamedInt4	TipcMsgUpdateNamedInt4	
updateNamedInt4Array	TipcMsgUpdateNamedInt4Array	
updateNamedInt8	TipcMsgUpdateNamedInt8	
updateNamedInt8Array	TipcMsgUpdateNamedInt8Array	
updateNamedMsg	TipcMsgUpdateNamedMsg	
updateNamedMsgArray	TipcMsgUpdateNamedMsgArray	
updateNamedMsgId		No parallel C function exists.
updateNamedMsgIdArray		No parallel C function exists.
updateNamedObject		No parallel C function exists.

Java Method Name	C Function Name	Comments and Exceptions
updateNamedReal4	TipcMsgUpdateNamedReal4	
updateNamedReal4Array	TipcMsgUpdateNamedReal4Array	
updateNamedReal8	TipcMsgUpdateNamedReal8	
updateNamedReal8Array	TipcMsgUpdateNamedReal8Array	
updateNamedStr	TipcMsgUpdateNamedStr	
updateNamedStrArray	TipcMsgUpdateNamedStrArray	
updateNamedTimestamp	TipcMsgUpdateNamedTimestamp	
updateNamedTimestamp Array	TipcMsgUpdateNamedTimestamp Array	
updateNamedUtf8	TipcMsgUpdateNamedUtf8	
updateNamedUtf8Array	TipcMsgUpdateNamedUtf8Array	
updateNamedXml	TipcMsgUpdateNamedXml	

Table 11 Interface TipcMt

Java Method Name	C Function Name	Comments and Exceptions
destroy	TipcMtDestroy	
getCompression	TipcMtGetCompression	
getDeliveryMode	TipcMtGetDeliveryMode	
getDeliveryTimeout	TipcMtGetDeliveryTimeout	
getGrammar	TipcMtGetGrammar	
getLbMode	TipcMtGetLbMode	
getLocalDelivery		No parallel C function exists.
getName	TipcMtGetName	
getNum	TipcMtGetNum	
getPriority	TipcMtGetPriority	
getUserProp	TipcMtGetUserProp	
setCompression	TipcMtSetCompression	
setDeliveryMode	TipcMtSetDeliveryMode	
setDeliveryTimeout	TipcMtSetDeliveryTimeout	
setLbMode	TipcMtSetLbMode	
setLocalDelivery		No parallel C function exists.
setPriority	TipcMtSetPriority	
setPriorityUnknown	TipcMtSetPriorityUnknown	
setUserProp	TipcMtSetUserProp	

Table 12 Interface TipcSrv

Java Method Name	C Function Name	Comments and Exceptions
addCreateCb	TipcSrvCreateCbCreate	
addDefaultCb	TipcSrvDefaultCbCreate	Inherited from the TipcConn class.
addDestroyCb	TipcSrvDestroyCbCreate	
addErrorCb	TipcSrvErrorCbCreate	Inherited from the TipcConn class.
addProcessCb	TipcSrvProcessCbCreate	
addProcessCb	TipcSrvProcessCbCreate	Inherited from the TipcConn class.
addQueueCb	TipcSrvQueueCbCreate	Inherited from the TipcConn class.
addReadCb	TipcSrvReadCbCreate	Inherited from the TipcConn class.
addWriteCb	TipcSrvWriteCbCreate	Inherited from the TipcConn class.
check	TipcSrvCheck	
create	TipcSrvCreate	
destroy	TipcSrvDestroy	
destroy	TipcSrvDestroy	Inherited from the TipcConn class.
flush	TipcSrvFlush	
getArch	TipcSrvGetArch	Inherited from the TipcConn class.
getAutoFlushSize	TipcSrvGetAutoFlushSize	Inherited from the TipcConn class.
getBlockMode	TipcSrvGetBlockMode	Inherited from the TipcConn class.
getConnStatus	TipcSrvGetConnStatus	
getDefaultControlCb		Inherited from the TipcConn class.
getDefaultErrorCb	TipcSrvErrorCbLookup	
getDefaultGmdFailureCb		Inherited from the TipcConn class.
getGmdDir	TipcGetGmdDir	Inherited from the TipcConn class.

Java Method Name	C Function Name	Comments and Exceptions
getGmdNumPending	TipcSrvGetGmdNumPending	Inherited from the TipcConn class.
getNode	TipcSrvGetNode	Inherited from the TipcConn class.
getNumQueued	TipcSrvGetNumQueued	Inherited from the TipcConn class.
getOption		Inherited from the TipcConn class.
getOptionBool		Inherited from the TipcConn class.
getOptionDouble		Inherited from the TipcConn class.
getOptionInt		Inherited from the TipcConn class.
getOptionStr		Inherited from the TipcConn class.
getPid	TipcSrvGetPid	Inherited from the TipcConn class.
getProperties		Inherited from the TipcConn class.
getQueueSize		Inherited from the TipcConn class.
getReadBufferSize	TipcSrvBufferGetReadSize	Inherited from the TipcConn class.
getServerLCN		No parallel C function exists.
getServerName		No parallel C function exists.
getSubjectLb	TipcSrvSubjectGetSubscribeLb	
getSubjectSubscribe	TipcSrvSubjectGetSubscribe	
getSubscribedList	TipcSrvSubjectTraverseSubscribe	
getTimeout	TipcSrvGetTimeout	Inherited from the TipcConn class.
getTrafficBytesRecv	TipcSrvTrafficGetBytesRecv	Inherited from the TipcConn class.
getTrafficBytesSent	TipcSrvTrafficGetBytesSent	Inherited from the TipcConn class.
getTrafficMsgsRecv	TipcSrvTrafficGetMsgsRecv	Inherited from the TipcConn class.
getTrafficMsgsSent	TipcSrvTrafficGetMsgsSent	Inherited from the TipcConn class.
getUniqueSubject	TipcSrvGetUniqueSubject	

Java Method Name	C Function Name	Comments and Exceptions
getUser	TipcSrvGetUser	Inherited from the TipcConn class.
getWriteBufferSize	TipcSrvBufferGetWriteSize	Inherited from the TipcConn class.
gmdFileDelete	TipcSrvGmdFileDelete	Inherited from the TipcConn class.
gmdMsgAck		Inherited from the TipcConn class.
gmdMsgDelete	TipcSrvGmdMsgDelete	Inherited from the TipcConn class.
gmdMsgResend		Inherited from the TipcConn class.
gmdMsgServerDelete	TipcSrvGmdMsgServerDelete	
gmdResend		Inherited from the TipcConn class.
insert	TipcSrvMsgInsert	Inherited from the TipcConn class.
isRunning	TipcSrvIsRunning	In Java, isRunning creates a connection if one does not already exist. In C, TipcSrvIsRunning creates and then destroys the connection.
keepAlive	TipcSrvKeepAlive	Inherited from the TipcConn class.
loadOptionsFile		Inherited from the TipcConn class.
loadOptionsStream		Inherited from the TipcConn class.
loadOptionsURL		Inherited from the TipcConn class.
logAddMt	TipcSrvLogAddMt	
logRemoveMt	TipcSrvLogRemoveMt	
lookupCreateCb	TipcSrvCreateCbLookup	
lookupDefaultCb	TipcSrvDefaultCbLookup	Inherited from the TipcConn class.
lookupDestroyCb	TipcSrvDestroyCbLookup	
lookupErrorCb	TipcSrvErrorCbLookup	Inherited from the TipcConn class.
lookupProcessCb	TipcSrvProcessCbLookup	

Java Method Name	C Function Name	Comments and Exceptions
lookupProcessCb	TipcSrvProcessCbLookup	Inherited from the TipcConn class.
lookupQueueCb	TipcSrvQueueCbLookup	Inherited from the TipcConn class.
lookupReadCb	TipcSrvReadCbLookup	Inherited from the TipcConn class.
lookupWriteCb	TipcSrvWriteCbLookup	Inherited from the TipcConn class.
mainLoop	TipcSrvMainLoop	Inherited from the TipcConn class.
makeSubjectAbsolute		Inherited from the TipcConn class.
next	TipcSrvMsgNext	
process	TipcSrvMsgProcess	Inherited from the TipcConn class.
read	TipcSrvRead	Inherited from the TipcConn class.
removeCreateCb		No parallel C function exists.
removeDefaultCb		Inherited from the TipcConn class.
removeDestroyCb		No parallel C function exists.
removeErrorCb		Inherited from the TipcConn class.
removeProcessCb		Inherited from the TipcConn class.
removeQueueCb		Inherited from the TipcConn class.
removeReadCb		Inherited from the TipcConn class.
removeWriteCb		Inherited from the TipcConn class.
search	TipcSrvMsgSearch	Inherited from the TipcConn class.
searchType	TipcSrvMsgSearchType	Inherited from the TipcConn class.
send	TipcSrvMsgSend	
sendRpc	TipcSrvMsgSendRpc	Inherited from the TipcConn class.
setAutoFlushSize	TipcSrvSetAutoFlushSize	Inherited from the TipcConn class.
setOption		Inherited from the TipcConn class.

Java Method Name	C Function Name	Comments and Exceptions
setSubjectSubscribe	TipcSrvSubjectSetSubscribe	
setSubjectSubscribeEx	TipcSrvSubjectSetSubscribeEx	
setTimeout	TipcSrvSetTimeout	Inherited from the TipcConn class.
setUsernamePassword	TipcSrvSetUsernamePassword	

Table 13 C Functions With No Java Equivalent

C Function Name	C Function Name
TipcBufMsgAppend	TipcBufMsgNext
TIpcCbConnProcessGmdFailure	TipcCbConnProcessKeepAliveCall
TipcCbSrvError	TipcCbSrvProcessControl
TipcCbSrvProcessGmdFailure	TipcConnCreate
TipcConnCreateClient	TipcConnCreateServer
TipcConnDecodeCbCreate	TipcConnDecodeCbLookup
TipcConnEncodeCbCreate	TipcConnEncodeCbLookup
TipcConnGetGmdMaxSize	TipcConnGetSocket
TipcConnGetXtSource	TipcConnGmdFileCreate
TipcConnLock	TipcConnMsgWrite
TipcConnMsgWriteVa	TipcConnSetGmdMaxSize
TipcConnSetSocket	TipcConnUnlock
TipcDeliveryModeToStr	TipcDispatcherSrvAdd
TipcDispatcherSrvRemove	TipcFtToStr
TipcInitThreads	TipcLbModeToStr
TipcMonExtSetReal16	TipcMonExtSetReal16Array
TipcMonPrintWatch	TipcMsgAck

C Function Name	C Function Name
TipcMsgAddNamedBinaryPtr	TipcMsgAddNamedBoolArrayPtr
TipcMsgAddNamedInt2ArrayPtr	TipcMsgAddNamedInt4ArrayPtr
TipcMsgAddNamedInt8ArrayPtr	TipcMsgAddNamedMsgArrayPtr
TipcMsgAddNamedMsgPtr	TipcMsgAddNamedReal4ArrayPtr
TipcMsgAddNamedReal8ArrayPtr	TipcMsgAddNamedReal16
TipcMsgAddNamedReal16Array	TipcMsgAddNamedReal16ArrayPtr
TipcMsgAddNamedStrArrayPtr	TipcMsgAddNamedStrPtr
TipcMsgAddNamedUtf8ArrayPtr	TipcMsgAddNamedUtf8Ptr
TipcMsgAddNamedXmlPtr	TipcMsgAppendBinaryPtr
TipcMsgAppendBoolArrayPtr	TipcMsgAppendInt2ArrayPtr
TipcMsgAppendInt4ArrayPtr	TipcMsgAppendInt8ArrayPtr
TipcMsgAppendMsgArrayPtr	TipcMsgAppendMsgPtr
TipcMsgAppendReal4ArrayPtr	TipcMsgAppendReal8ArrayPtr
TipcMsgAppendReal16	TipcMsgAppendReal16Array
TipcMsgAppendReal16ArrayPtr	TipcMsgAppendStrArrayPtr
TipcMsgAppendStrPtr	TipcMsgAppendStrAReal8
TipcMsgAppendUtf8ArrayPtr	TipcMsgAppendUtf8Ptr
TipcMsgAppendXmlPtr	TipcMsgCreate
TipcMsgDestroy	TipcMsgFieldUpdateBinaryPtr
TipcMsgFieldUpdateBoolArrayPtr	TipcMsgFieldUpdateInt2ArrayPtr
TipcMsgFieldUpdateInt4ArrayPtr	TipcMsgFieldUpdateInt8ArrayPtr
TipcMsgFieldUpdateReal4ArrayPtr	TipcMsgFieldUpdateReal8ArrayPtr
TipcMsgFieldUpdateReal16ArrayPtr	TipcMsgFieldUpdateStrPtr

C Function Name	C Function Name
TipcMsgFieldUpdateTimestampArrayPtr	TipcMsgFieldUpdateUtf8Ptr
TipcMsgFieldUpdateXmlPtr	TicpMsgFieldSetSize
TipcMsgFileCreate	TipcMsgFileCreateFromFile
TipcMsgFileDestroy	TipcMsgFileRead
TipcMsgFileWrite	TipcMsgGetDeliveryMode
TipcMsgGetHeaderStrEncode	TipcMsgGetNamedReal16
TipcMsgGetNamedReal16Array	TipcMsgGetRefCount
TIpcMsgIncrRefCount	TipcMsgNextReal16
TipcMsgNextReal16Array	TipcMsgNextStrReal8
TipcMsgPrintError	TipcMsgRead
TipcMsgReadVa	TipcMsgSetHeaderStrEncode
TipcMsgTraverse	TipcMsgUpdateNamedBinaryPtr
TipcMsgUpdateNamedBoolArrayPtr	TipcMsgUpdateNamedInt2ArrayPtr
TipcMsgUpdateNamedInt4ArrayPtr	TipcMsgUpdateNamedInt8ArrayPtr
TipcMsgUpdateNamedMsgArrayPtr	TipcMsgUpdateNamedMsgPtr
TipcMsgUpdateNamedReal4ArrayPtr	TipcMsgUpdateNamedReal8ArrayPtr
TipcMsgUpdateNamedReal16	TipcMsgUpdateNamedReal16Array
TipcMsgUpdateNamedReal16ArrayPtr	TipcMsgUpdateNamedStrArrayPtr
TipcMsgUpdateNamedStrPtr	TipcMsgUpdateNamedUnknown
TipcMsgUpdateNamedUtf8ArrayPtr	TipcMsgUpdateNamedUtf8Ptr
TipcMsgUpdateNamedXmlPtr	TipcMsgWriteVa
TipcMtCreate	TipcMtGetHeaderStrEncode
TipcMtLogAddMt	TipcMtLogRemoveMt

C Function Name	C Function Name
TipcMtLookup	TipcMtLookupByNum
TipcMtPrint	TipcMtSetHeaderStrEncode
TipcMtTraverse	TipcPropertiesCreate
TipcPropertiesCreateMsg	TipcPropertiesDestroy
TipcPropertiesGet	TipcPropertiesGetCount
TipcPropertiesGetDefault	TipcPropertiesMsgCreate
TipcPropertiesSet	TipcPropertiesTraverse
TipcSrvGetGmdMaxSize	TipcSrvGetSocket
TipcSrvGetXtSource	TipcSrvGmdFileCreate
TipcSrvGmdMsgStatus	TipcSrvLock
TipcSrvMsgWrite	TipcSrvMsgWriteVa
TipcSrvPrint	TipcSrvSetGmdMazSize
TipcSrvSetSocket	TipcSrvStdSubjectSetSubscribe
TipcSrvStop	TipcSrvSubjectCbCreate
TipcSrvSubjectCbDestroyAll	TipcSrvSubjectCbLookup
TipcSrvSubjectDefaultCbCreate	TipcSrvSubjectDefaultCbLookup
TipcSrvSubjectGmdInit	TipcSrvSubjectLbInit
TipcSrvTraverseCbCreate	TipcSrvTraverseCbLookup
TipcSrvUnlock	TipcStrToDeliveryMode
TipcStrToFt	TipcStrToLbMode

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

T

U

W