

# **TIBCO SmartSockets™**

## **C++ User's Guide**

*Software Release 6.8  
July 2006*

## Important Information

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

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

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

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

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

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

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

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

TIBCO Software Inc. Confidential Information

# Contents

<b>Preface</b> .....	<b>v</b>
Intended Audience .....	vi
Related Documentation .....	vii
TIBCO Product Documentation .....	vii
Using the Online Documentation .....	vii
Conventions Used in This Manual .....	viii
Typeface Conventions .....	viii
Notational Conventions .....	ix
Identifiers .....	ix
Case .....	x
How to Contact TIBCO Support .....	xi
 <b>Chapter 1 Introduction to the C++ Class Library</b> .....	 <b>1</b>
C++ Class Library Overview .....	2
C++ Class Library Features .....	2
C++ Namespace .....	2
Native Type Names .....	3
Exception Handling .....	3
Constant Objects .....	4
Callback Support .....	4
Utilities Function Wrappers .....	4
Multiple Connection Support .....	7
 <b>Chapter 2 Using the C++ Class Library</b> .....	 <b>9</b>
Example: C++ TIBCO SmartSockets Application .....	10
The Sender Program .....	10
The Receiver Program .....	12
Compiling, Linking, and Running .....	15
Using Callbacks to Process Messages .....	17
Exception Handling .....	20
Exception Class Hierarchy .....	21
Exception Class Features .....	22
Include Files .....	22

Source File Distribution . . . . . 23

    The Binary Library . . . . . 23

    The Source Code . . . . . 23

    Source File Organization . . . . . 24

Using Threads . . . . . 26

**Index . . . . . 27**

# 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

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

This guide describes the C++ class interface for the SmartSockets library. It also provides detailed descriptions and examples of how to set up, compile and run C++ applications using the C++ class library.

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

## Topics

---

- *Intended Audience, page vi*
- *Related Documentation, page vii*
- *Conventions Used in This Manual, page viii*
- *How to Contact TIBCO Support, page xi*

## Intended Audience

---

This guide is intended for C++ programmers who plan to develop with SmartSockets in an object-oriented manner, without using the C application programming interface (API).

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

- working knowledge of C++
- familiarity with the operating system is required for developing SmartSockets applications (UNIX, Windows, or whatever platform is running SmartSockets). This includes knowing how to log in, log out, edit a text file, change directories, list files, and compile, link, and run a program.
- understand general messaging and publish/subscribe concepts and terminology
- familiarity with the SmartSockets messaging concepts covered in the *TIBCO SmartSockets User's Guide*.

## Related Documentation

---

This section lists documentation resources you may find useful.

### TIBCO Product Documentation

The following documents form the SmartSockets documentation set:

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

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

## Notational Conventions

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

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

## Identifiers

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

These are examples of valid identifiers:

```
EPS
battery_11
K11
_
_all
```

These are invalid identifiers:

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

## Case

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

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

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

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

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

## How to Contact TIBCO Support

---

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

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

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

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

<http://support.tibco.com>

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



# Introduction to the C++ Class Library

SmartSockets Versions 6.2 and above include the `ssc++` class library. The `ssc++` library is the preferred C++ class library for SmartSockets development. This chapter provides an overview of the `ssc++` class library features.

Prior to Version 6.2, the only C++ library included with SmartSockets was the `cxxipc` library. Although the old `cxxipc` library is maintained for backwards compatibility, it does not include new SmartSockets features. For more information on the `cxxipc` library, see the *TIBCO SmartSockets cxxipc Class Library*.

All new SmartSockets C++ development should use the `ssc++` library. Throughout the rest of this book, the `ssc++` library is referred to as the C++ class library.

The C++ class library includes:

- a namespace for SmartSockets
- support for multiple RTserver connections
- a model for callbacks that does not require static callback methods
- an exception hierarchy that shows where errors occurred
- wrappers for select C utilities functions
- use of native types

The C++ class library reference documentation is available in HTML format. Access the reference material from the HTML documentation interface. See the *TIBCO SmartSockets cxxipc Class Library* for the `cxxipc` library.

## Topics

---

- *C++ Class Library Overview, page 2*
- *C++ Class Library Features, page 2*

## C++ Class Library Overview

---

When working in C++, you interact with the SmartSockets C++ class library in these ways:

- by constructing a C++ object using an appropriate class constructor, then invoking the member functions of that object
- by invoking static member functions of a class without necessarily constructing any particular objects of that class
- by deriving new C++ classes as subtypes of provided classes to extend or otherwise modify the behavior of SmartSockets classes

This chapter describes some features of the SmartSockets C++ class library. Additional reference material is available in HTML format. Access the reference material from the HTML documentation interface.

The *TIBCO SmartSockets User's Guide*, *TIBCO SmartSockets Application Programming Interface*, and the *TIBCO SmartSockets Utilities*, which are companion reference documents to this manual, describe RTserver and all other features using the C programming language API. Most SmartSockets class member functions are simple C++ wrappers around a corresponding C language API function.

## C++ Class Library Features

---

This library provides many new features not included in the `cxxtipc` library, including the SmartSockets namespace, use of native type names, improved exception and callback handling, wrappers for utilities functions, and support for multiple RTserver connections.

### C++ Namespace

The C++ class library uses the SmartSockets namespace to simplify class names. If there are conflicts between the SmartSockets C++ library and another library, prefix SmartSockets class names with `SmartSockets::`. For example, `TipcConn` can also be written as `SmartSockets::TipcConn`.

## Native Type Names

This is a mapping of SmartSockets types to native C++ types:

Type	Declared as:
T_INT1	int1
T_UINT1	uint1
T_INT2	int2
T_UINT2	uint2
T_INT4	int4
T_UINT4	uint4
T_INT8	int8
T_REAL4	real4
T_REAL8	real8
T_REAL16	real16

Wherever possible, these names are also changed:

Name	Changed to:
T_STR	char *
T_PTR	void *
T_BOOL	bool

## Exception Handling

The Exception class is used to categorize errors. The C++ class library explicitly declares which exceptions are thrown from each class. The exception hierarchy follows the main classes in the class hierarchy. For example, exceptions generated from methods in the TipcMt class throw TipcMtException exceptions. See Exception Handling on page 20 for more information on the Exception class.

## Constant Objects

The C++ class library uses the `const` indicator for all methods that do not modify the contents of an object. For example, methods that received a literal string in the `cxxtipc` library now receive `const char *`.

## Callback Support

Callback objects can now be passed as parameters into registration methods. If registration is successful, the registration method returns a pointer to the callback class and a template parameter showing the type of callback used. The callback classes are:

- `ConnectionCallback`
- `ErrorCallback`
- `DecodeCallback`
- `MessageCallback`
- `EncodeCallback`
- `MessageQueueCallback`

## Utilities Function Wrappers

The C++ class library includes support for some Tut\* functions from the C API. These methods are in four new classes:

- `System` — static methods that access the operating system
- `Utilities` — static methods that access the SmartSockets application
- `Option` — methods that create and configure options
- `XML` — methods that manipulate XML objects

The classes and their methods, with their corresponding C API functions, are shown in Table 1.

Table 1 Utilities Function Wrappers

Class	Method Name	C Function Name
<b>System</b>		
	<code>exit</code>	<code>TutExit</code>
	<code>getFloatingPointFormat</code>	<code>TutGetRealFormat</code>
	<code>getIntFormat</code>	<code>TutGetIntFormat</code>
	<code>getTime</code>	<code>TutGetWalltime</code>

Table 1 Utilities Function Wrappers

Class	Method Name	C Function Name
	runCommand	TutSystem
	sleep	TutSleep
<b>Utilities</b>		
	getSocketDir	TutGetSocketDir
	getVersionName	TutGetVersionName
	getVersionNumber	TutGetVersionNumber
	parseCommandFile	TutCommandParseFile
	parseCommandString	TutCommandParseString
	parseTypedCommandString	TutCommandParseTypedString
<b>Option</b>		
	create	TutOptionCreate
	destroyOption	TutOptionDestroy
	getBool	TutOptionGetBool
	getEnum	TutOptionGetEnum
	getEnumList	TutOptionGetEnumList
	getName	TutOptionGetName
	getReadOnly	TutOptionGetReadOnly
	getReal8	TutOptionGetNum
	getRequired	TutOptionGetRequired
	getString	TutOptionGetStr
	getStringList	TutOptionGetStrList
	getType	TutOptionGetType
	isKnown	TutOptionGetKnown

Table 1 Utilities Function Wrappers

Class	Method Name	C Function Name
	option constructor	TutOptionLookup
	optionChangeCbCreate	TutOptionChangeCbCreate
	setBool	TutOptionSetBool
	setEnum	TutOptionSetEnum
	setEnumList	TutOptionSetEnumList
	setReadOnly	TutOptionSetReadOnly
	setReal8	TutOptionSetNum
	setRequired	TutOptionSetRequired
	setString	TutOptionSetStr
	setStringList	TutOptionSetStrList
	setUnKnown	TutOptionSetUnknown
XML		
	clone	TutXmlClone
	createFromStaticBuf	TutXmlCreateStatic
	getString	TutXmlGetStr
	option constructor	TutXmlCreate
	setString	TutXmlSetStr

## Multiple Connection Support

The C++ class library supports multiple connections, allowing an RTclient to connect to multiple RTservers. Multiple connections are supported by TipcSrv class, in conjunction with these methods and classes:

- **setCredentials** — the TipcSrv class includes the method setCredentials, which sets credentials for the connection. This allows you to use any credential mechanism, such as a certificate or user name and password, to authenticate a client application before it joins the RTserver cloud.
- **TipcSrvCache** — the TipcSrvCache class allows you to store messages in a memory cache. Use the setSubscribeCache method in the TipcSrv class to enable caching.
- **TipcDispatcher** — the TipcDispatcher class creates RTclient dispatchers. A dispatcher manages events and incoming messages. For more information on dispatchers, see the *TIBCO SmartSockets User's Guide*.
- **TipcEvent** — the TipcEvent class creates events. Events are objects registered in a dispatcher. There are five kinds of events: connection, message, socket, timer, and user. For more information on events, see the *TIBCO SmartSockets User's Guide*.
- **TipcMon** — The TipcMon class works with the TipcSrv class to monitor various aspects of a SmartSockets application. In RTclients using multiple connections, a reference to the connection where monitoring operations are carried out is a necessary parameter.

Extension data from RTclients, which is data created within an RTclient, can be monitored by another RTclient with the TipcMonExt\* or TipcSrvMonExt\* APIs. The RTserver is not involved in generating this kind of monitoring information.

The C++ class library has the same advantages and limitations as the SmartSockets multiple connections API. Some SmartSockets commands, such as connect, disconnect, subscribe, and unsubscribe, cannot be used with C++.



## Chapter 2      **Using the C++ Class Library**

This chapter introduces a simple C++ TIBCO SmartSockets application using the C++ class library. The mechanics of compiling and linking applications using the class library are also included.

### Topics

---

- *Example: C++ TIBCO SmartSockets Application, page 10*
- *Exception Handling, page 20*
- *Include Files, page 22*
- *Source File Distribution, page 23*
- *Using Threads, page 26*

## Example: C++ TIBCO SmartSockets Application

---

These two C++ language example programs illustrate a simple SmartSockets application that uses the C++ class library. The first program, the sender program, uses the RTserver to publish a message consisting of two strings to the receiver program.

The files needed to compile, link, and run the example programs are located in this directory:

### UNIX:

`$RTHOME/examples/sscpp`

### Windows:

`%RTHOME%\examples\sscpp`

## The Sender Program

The sender example is a program written in C++ that uses the RTserver to publish two strings in the message to another program. A discussion of the highlights follows the program example.

```

1  #include <rtworks/sscpp.h>
2  using namespace SmartSockets;
3  int main(int argc, char **argv)
4  {
5      TipcSrv *srv;
6
7      // Set the name of the project.
8      try {
9          Option opt("project");
10         opt.setEnum("ipc_example");
11     }
12     catch (OptionException oe) {
13         Utilities::out("Error creating options: %s\n", oe.what());
14         return T_EXIT_FAILURE;
15     }
16
17     // Connect to RTserver, and open the connection.
18     try {
19         srv = new TipcSrv("", NULL, NULL, NULL);
20         srv->open();
21     }
22     catch (TipcSrvException se) {
23         Utilities::out("Error in creating connection: %s\n", se.what());
24         return T_EXIT_FAILURE;
25     }
26 }
```

```

17 // Create a message.
18 TipcMsg msg(T_MT_STRING_DATA);
19 try {
20     msg.setDest((const char*)"demo");
21
22     // Build the message with two string fields: "x" and "Hello World".
23     msg << (const char*)"x" << (const char*)"Hello World";
24     if (!msg) {
25         Utilities::out("Error appending fields of TipcMsg object.\n");
26         return T_EXIT_FAILURE;
27     }
28 }
29 catch (TipcMsgException me) {
30     Utilities::out("Error in creating and appending message: %s\n", me.what());
31     return T_EXIT_FAILURE;
32 }
33
34 // Send the message.
35 try {
36     srv->send(msg);
37     srv->flush();
38     srv->close();
39 }
40 catch (TipcSrvException se) {
41     Utilities::out("Error in sending message through the connection: %s\n",
42                   se.what());
43     return T_EXIT_FAILURE;
44 }
45
46 return T_EXIT_SUCCESS;
47 } //main

```

Some things to notice about the sender program:

- Line 1      The first line of the program, `#include <rtworks/sscpp.h>`, must be included in every program that uses the SmartSockets C++ class library. It contains a series of `#include` statements for the various header files of the class library.
- Line 2      Sets the SmartSockets namespace. The SmartSockets namespace prevents naming conflicts between SmartSockets and other applications.
- Lines 6-7   The Option constructor creates a project, and the `setEnum` method sets project name for the sender program. The receiver program uses the same project name. See the *TIBCO SmartSockets User's Guide* for more information on project names.
- Lines 12-13 Obtains a handle to a `TipcSrv` object, then creates and opens a connection to RTserver. By default, this creates a full connection.

- Lines 17-19 Constructs a `TipcMsg` object by passing the `T_MT_STRING_DATA` message type as an argument to the constructor. The message destination is set to `demo`. See the *TIBCO SmartSockets User's Guide* for more information on the destination property of messages.
- Line 20 Appends data fields to the `TipcMsg` object. This example illustrates a means of appending data that is unique to the C++ class library as compared to the C API. Data is appended by using overloaded insertion operators in a function chain. Data can also be appended using the overloaded `TipcMsg::append` member function, which provides an interface similar to the C API `TipcMsgAppend*` functions.
- Lines 28-30 Publishes the message and closes the connection to `RTserver` by calling the `TipcSrv` member functions `TipcSrv::send`, `TipcSrv::flush`, and `TipcSrv::close`.

## The Receiver Program

The receiver example is a program written in C++ that uses the `RTserver` to receive two strings in the message from another program. A discussion of the highlights follows the program example.

```

1  #include <rtworks/sscpp.h>
2  using namespace SmartSockets;
3  int main(int argc, char **argv)
4  {
5      TipcSrv *srv;
6
7      // Set the name of the project.
8      try {
9          Option opt("project");
10         opt.setEnum("ipc_example");
11     }
12     catch (OptionException oe) {
13         Utilities::out("Error creating options: %s\n", oe.what());
14         return T_EXIT_FAILURE;
15     }
16
17     // Connect to RTserver, and open the connection.
18     try {
19         srv = new TipcSrv("", NULL, NULL, NULL);
20         srv->open();
21
22         // Subscribe to receive any messages published to the "demo" subject.
23         srv->setSubscribe((const char*)"demo");
24     }
25 }
```

```

15     catch (TipcSrvException se)
16     {
17         Utilities::out("Error in creating connection: %s\n", se.what());
18         return T_EXIT_FAILURE;
19     }

18     const char *var_name;
19     const char *var_value;

    // Get the next incoming message.
20     try {
21         TipcMsg msg;
22         if(!srv->nextEx(msg, T_TIMEOUT_FOREVER))
23         {
24             Utilities::out("Timeout reached.\n");
25             return T_EXIT_FAILURE;
26         }

27         if (!msg) {
28             Utilities::out("Error creating TipcMsg object.\n");
29             return T_EXIT_FAILURE;
30         }

    // Set the pointer to first field in message
31         msg.setCurrent(0);

    // Extract the information from the received message.
32         msg >> var_name >> var_value;
33         if (!msg) {
34             Utilities::out("Error reading fields of TipcMsg object.\n");
35             return T_EXIT_FAILURE;
36         }
37     }
38     catch(TipcMsgException me) {
39         Utilities::out("Error in TipcMsg class: %s\n", me.what());
40         return T_EXIT_FAILURE;
41     }

    // Display the values on stdout.
42     Utilities::out("Variable Name = %s\n", var_name);
43     Utilities::out("Variable Value = %s\n", var_value);

44     srv->close();
45     return T_EXIT_SUCCESS;
46 } //main

```

Some things to notice about the receiver program:

- Line 1      The first line of the program, `#include <rtworks/sscpp.h>`, must be included in every program that uses the SmartSockets C++ class library. It contains a series of `#include` statements for the various header files of the class library.
- Line 2      Sets the SmartSockets namespace. The SmartSockets namespace prevents naming conflicts between SmartSockets and other applications.
- Lines 6-7   The Option constructor creates a project, and the `setEnum` method sets project name for the receiver program. The sender program uses the same project name.
- Lines 12-13 Obtains a handle to a `TipcSrv` object, then creates and opens a connection to `RTserver`.
- Line 14      Subscribes to the subject, `demo`, by calling the `TipcSrv::setSubscribe` member function. Note that the sender program designated `demo` as the destination of the message. See the *TIBCO SmartSockets User's Guide* for information about subjects.
- Lines 21-22 Creates a `TipcMsg` object and calls the `TipcSrv::nextEx` member function, with `T_TIMEOUT_FOREVER` and the `TipcMsg` object as arguments to the call.  
  
The result of the `nextEx` member function is the `TipcMsg` object or a `NULL`.
- Line 28      After receiving a message from `RTserver`, the receiver program sets the pointer to first field in message with the `TipcMsg::setCurrent` member function.
- Line 29      Extracts the data fields from the message with a C++ function chain of overloaded extraction operators. Data are also extracted using the overloaded `TipcMsg::next` member functions, which provide an interface similar to the C API `TipcMsgNext*` functions.
- Lines 36-37 Displays the values of the data fields by outputting the data using the `Utilities::out` method.



This example avoids using `cout` from `IOSTREAM`. Some SmartSockets functions use `Utilities::out` to display status information. Because `Utilities::out` uses the `stdout` stream and `cout` typically uses its own output stream, use `Utilities::out` in `RTclient` programs, rather than `cout`, so that output prints in sequential order.

## Compiling, Linking, and Running

Follow these steps to compile, link, and run the sender and receiver example programs.

### Step 1 **Copy the `sndr.cxx` and `rcvr.cxx` programs**

To compile, link, and run the example programs, you must copy the programs to your own directory. The programs are located in this directory:

#### **UNIX:**

```
$RTHOME/examples/sscpp
```

#### **Windows:**

```
%RTHOME%\examples\sscpp
```

### Step 2 **Start the RTserver**

If RTserver is not already running, start it:

#### **UNIX:**

```
$ rtserver
```

#### **Windows:**

```
$ 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 3 **Compile and link the sender and receiver programs**

Use these commands to compile and link the programs:

#### **UNIX:**

```
$ rtlink -cpp -o sndr.x sndr.cxx
$ rtlink -cpp -o rcvr.x rcvr.cxx
```

#### **Windows:**

```
$ nmake /f sndrw32m.mak
$ nmake /f rcvrw32m.mak
```

On a UNIX system the `rtlink` command by default uses the C compiler `cc` command to compile and link. Specifying the `-cpp` flag tells `rtlink` to use the native C++ compiler (for example, on a Solaris platform the compiler is named `CC`) and adds the SmartSockets C++ class library to the list of libraries to be linked into the executable. To use a C++ compiler other than the default compiler, set the environment variable `CC` to the name of the compiler. `rtlink` then uses this compiler.

For example, these commands are used to compile and link on UNIX with the GNU C++ compiler `g++`:

**UNIX:**

```
$ env CC=g++ rtlink -cpp -o sndr.x sndr.cxx
$ env CC=g++ rtlink -cpp -o rcvr.x rcvr.cxx
```

**Step 4 Start the receiver program**

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

Start up the receiving program in the first window:

**UNIX:**

```
$ rcvr.x
```

**Windows:**

```
$ rcvr.exe
```

**Step 5 Start the sender program**

In a separate window from where the receiving program is running, start up the sending program:

**UNIX:**

```
$ sndr.x
```

**Windows:**

```
$ sndr.exe
```

## Program Output

The output from the sending process is similar to this:

```
Connecting to project <ipc_example> on <_node> RTserver
Using tcp protocol
Message from RTserver: Connection established.
Start subscribing to subject </_workstation1_6607>
```

The output from the receiving process is similar to this:

```
Connecting to project <ipc_example> on <_node> RTserver
Using tcp protocol
Message from RTserver: Connection established.
Start subscribing to subject </_workstation1_6605>
Variable Name   = x
Variable Value  = Hello World
```

## Using Callbacks to Process Messages

The receiver program can also be written to use callbacks for message processing. The `rcvrcb.cpp` example program, like the receiver program, uses the RTserver to receive two strings in the message from another program. However, the `rcvrcb.cpp` program reads and processes the received message within a callback.

A discussion of the highlights follows the program example.

```
1  #include <rtworks/sscpp.h>
2
3  using namespace SmartSockets;
4
5  //=====
6  //..numcb -- numeric data callback
7
8  class msg_cb
9  : public MessageCallBack
10 {
11     public:
12
13     virtual void onMessage( Callback<MessageCallBack>* callback,
14                           TipcMsg & msg,
15                           TipcConn & conn)
16     {
17         Utilities::out("Entering msg_cb.\n");
18         const char *var_name;
19         const char *var_value;
20
21         try {
22             msg.setCurrent(0);
23             // Extract the information from the received message.
24             msg >> var_name >> var_value;
25             if (!msg) {
26                 Utilities::out("Error reading fields of TipcMsg object.\n");
27             }
28         }
29     }
30 }
```

```

15         catch(TipcMsgException me) {
16             Utilities::out("Error in TipMsg class: %s\n", me.what());
17         }

        // Display the values on stdout.
17         Utilities::out("Variable Name = %s\n", var_name);
18         Utilities::out("Variable Value = %s\n", var_value);
19     }
20 };

19 int main(int argc, char **argv)
20 {
21     TipcSrv *srv;
22     msg_cb *mcb = new msg_cb();
23     Callback<MessageCallback>* cb;

    // Set the name of the project.
23     try {
24         Option opt("project");
25         opt.setEnum("ipc_example");
26     }
27     catch (OptionException oe) {
28         Utilities::out("Error creating options: %s\n", oe.what());
29         return T_EXIT_FAILURE;
30     }

    // Connect to RTserver, and open the connection.
29     try {
30         srv = new TipcSrv("", NULL, NULL, NULL);
31         srv->open();

        // Subscribe to receive any messages published to the "demo" subject.
32         srv->setSubscribe((const char*)"demo");
33     }
34     catch (TipcSrvException se)
35     {
36         Utilities::out("Error in creating connection: %s\n", se.what());
37         return T_EXIT_FAILURE;
38     }

    // process callback for STRING_DATA
36     try {
37         TipcMt mt = TipcMt::lookup(T_MT_STRING_DATA);
38         Callback<MessageCallback>* cb = srv->processCbCreate(mt, mcb);
39     }
40     catch (TipcMtException mte) {
41         Utilities::out("Exception on mt lookup for STRING_DATA. %s\n", mte.what());
42         return T_EXIT_FAILURE;
43     }
44     catch (TipcSrvException srve) {
45         Utilities::out("Error in creating process callback. %s\n", srve.what());
46         return T_EXIT_FAILURE;
47     }

45     try {
46         srv->mainLoop(45.0);
47     }

```

```

47     catch (TipcSrvException se) {
48         if (se.getErrNum() != T_ERR_TIMEOUT_REACHED) {
49             Utilities::out("Server main loop failed.\n");
49         }
50     }
51     Utilities::out("Exception in mainloop. %s\n", se.what());
52 }
53
51     cb->destroy();
52     delete mcb;
53
53     srv->close();
54     return T_EXIT_SUCCESS;
54 } //main

```

Some things to notice about the `rcvrCb.cxx` program:

- Lines 3-18    The declaration of the callback structure. In the callback, messages are processed as they were in lines 28-37 of the receiver program, on page 12.
- Line 6        The `onMessage` method is the default callback message handler.
- Line 21       Creates a callback on `RTserver`.
- Lines 22-35   Creates a project and sets the project name, creates and opens a connection to `RTserver`, and subscribes to the subject, `demo`. These lines correspond to lines 5-17 in the receiver program, on page 12.
- Line 37       Creates a message type object of type `T_MT_STRING_DATA`.
- Line 38       Creates a callback using `processCbCreate`.
- Lines 39-44   Provides exception handling if the program cannot find the message type or cannot create the callback.
- Line 46       Uses the `mainLoop` method to check for messages for 45 seconds. Any messages received during that time are processed by the callback.
- Lines 51-52   Destroys the callback and frees up the memory.

## Exception Handling

---

Most of the SmartSockets functions return `FALSE` on failure, and `TRUE` on success. The corresponding C++ member functions throw exceptions on failure. Each class in the C++ library has an associated exception class.

A block written in C looks similar to this:

```
if (!TipcMsgAppendStr(msg, "voltage")) {
    TutOut("Could not append first field.\n");
    return T_EXIT_FAILURE;
}
```

In C++, this same code looks similar to this:

```
try {
    msg << "voltage";
}
catch (TipcMsgException msge) {
    TutOut("Could not append first field.\n");
    return T_EXIT_FAILURE;
}
```

In the next example, `TipcMsg::getDest` calls the C function `TipcMsgGetDest`. If the C API returns `FALSE`, then a `TipcMsgException` exception is thrown to the user:

```
/* ===== */
/*..TipcMsg::dest -- get the destination property of a message */

const char * TipcMsg::getDest() const throw (TipcMsgException)
{
    char * dest_return = (char *)""; // initialize variable

    if (!TipcMsgGetDest(_msg, &dest_return))
        throw TipcMsgException();

    return const_cast<const char*> (dest_return);
}
```

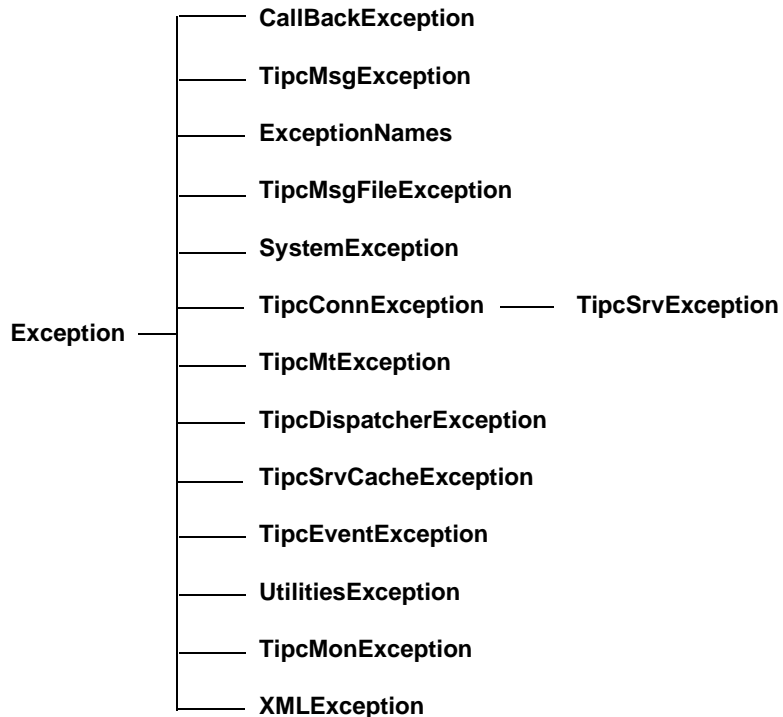
Use the `try/catch` block to handle unexpected behavior from the member function:

```
try {
    TipcMsg msg;
    char* the_dest = (char *)msg.getDest();
}
catch (TipcMsgException msge) {
    // handle the error
}
```

## Exception Class Hierarchy

Each main class in the C++ library has an associated Exception class. The exception hierarchy follows the main classes in the class hierarchy. For example, exceptions generated from methods in the TipcMt class throw TipcMtException exceptions. Figure 1 shows the Exception class hierarchy.

*Figure 1 Exception Class Inheritance Hierarchy*



## Exception Class Features

Each Exception class includes these member functions, which retrieve information about errors that are generated:

- `what` — retrieves the descriptive string associated with the SmartSockets error number
- `getErrNum` — retrieves the SmartSockets error number
- `getOSErrNum` — retrieves the error number for an operating system error
- `getSocketErrNum` — retrieves the error number for a socket error
- `getCErrNum` — retrieves the error number for a C error

For more information on all error codes, see the *TIBCO SmartSockets API Quick Reference*.

## Include Files

---

Each `.cxx` source file has a corresponding `.h` header file containing the declaration of a SmartSockets class. Code written in C++ that uses the C++ class library must include the header file `sscpp.h`, which is located in this directory:

### UNIX:

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

### Windows:

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

This include file automatically includes all the header files used for interprocess communication using the C++ class library.

## Source File Distribution

---

The C++ class library is distributed in binary and source code format.

### The Binary Library

The binary library, whose suffix may differ according to platform, is included with the SmartSockets distribution in these directories:

**UNIX:**

```
$RTHOME/lib/$RTARCH/librtsscnp50.so
```

**Windows:**

```
%RTHOME%\lib\%RTARCH%\rtsscnp.lib
```

The binaries are compiled with a native compiler from a vendor's platform. For example, a Sun SPARCCompiler was used to compile binaries for Solaris.

### The Source Code

Because C++ compilers do not necessarily generate binary code that is compatible with other C++ compilers, the C++ source is also distributed so that you can compile the class library with a C++ compiler compatible with your environment. The source files are located in this directory:

**UNIX:**

```
$RTHOME/source/sscnp
```

**Windows:**

```
%RTHOME%\source\sscnp
```

A sample makefile is included with the source files to build the C++ library. The library name produced and the name of the directory are:

**UNIX:**

```
$RTHOME/lib/$RTARCH/libsscnp.a
```

**Windows:**

```
%RTHOME%\lib\%RTARCH%\sscnp.lib
```

On UNIX, the sample makefile builds the library with the Sun WorkShop C++ compiler, CC.

This makefile does not overwrite the TIBCO library from the product distribution.

Source File Organization

The source file organization is:

Table 2 SmartSockets Source File Organization

Source File	SmartSockets C++ Class Implementation
tcallbck.cxx	CallBack
	ConnectionCallBack
	DecodeCallBack
	EncodeCallBack
	ErrorCallBack
	MessageCallBack
	MessageQueueCallBack
tconn.cxx	TipcConn
	TipcConnServer
	TipcConnClient
	TipcConnSearchSelector
tdisp.cxx	TipcDispatcher
	TipcDispatcherTraverser
tevent.cxx	TipcEvent
	ConnEvent
	MessageEvent
	SocketEvent
	TimeEvent

*Table 2 SmartSockets Source File Organization*

Source File	SmartSockets C++ Class Implementation
tex.cxx	Exception
	CallbackException
	SystemException
	TipcConnException
	TipcSrvException
	TipcDispatcherException
	TipcEventException
	TipcMonException
	TipcMsgException
	TipcMsgFileException
	TipcMtException
	TipcSrvCacheException
	UtilitiesException
	XMLException
texnames.cxx	ExceptionNames
tmon.cxx	TipcMon
tmsg.cxx	TipcMsg
	TipcMsgTraverser
tmsgfile.cxx	TipcMsgFile
tmsgname.cxx	TipcMsgManipName
tmsgsize.cxx	TipcMsgManipSize
tmt.cxx	TipcMt
	TipcMtTraverser
toption.cxx	Option
	OptionChangeCallBack

*Table 2 SmartSockets Source File Organization*

Source File	SmartSockets C++ Class Implementation
<code>tscache.cxx</code>	TipcSrvCache
<code>tsrv.cxx</code>	TipcSrv
<code>tsystem.cxx</code>	System
<code>tutil.cxx</code>	Utilities
<code>txml.cxx</code>	XML

## Using Threads

---

To use threads in your application, you must initialize them. Programs that call SmartSockets methods from more than one thread must first call `Utilities::initThreads`, even if they do not use any of the other thread methods. `Utilities::initThreads` initializes the thread API and turns on internal thread synchronization calls. This protects the integrity of the library's internal data structures.

If threads are used in a program, `Utilities::initThreads` must be called before any other SmartSockets method. This member function calls the `TipcInitThreads` function in C. See the *TIBCO SmartSockets Utilities* for more details on the threads API.

# Index

## C

- caching 7
- callbacks 4, 17
- case sensitivity x
  - on UNIX and Windows x
- connections
  - multiple connections 7
- const indicator 4
- constant objects 4
- cout
  - in SmartSockets programs 14
- credentials 7
- customer support xi

## D

- dispatchers 7
- documentation vii

## E

- error handling 3
- events 7
- exception handling 3
- extension data, monitoring 7

## F

- file names
  - specifying x

- functions

- case-sensitivity x
  - utility 4

## H

- header file
  - include files 22
- hierarchical namespace
  - see namespace
- HTML
  - documentation vii

## I

- identifiers
  - case sensitivity x
- include files
  - header file 22

## M

- memory cache 7
- messages
  - accessing data fields 14
  - appending data fields 12
  - case sensitivity x
  - processing using callbacks 17
- monitoring
  - RTclient data 7
  - SmartSockets data 7
- multiple connections 7

## N

- namespace
  - SmartSockets namespace 2
- native type names 3

## O

- options
  - case sensitivity x

## R

- RTclient extension data, monitoring 7
- rtserver64 command 15

## S

- shell commands
  - specifying x
- SmartSockets C++ Class Library
  - an example 10
- SmartSockets namespace 2
- source files
  - binary library 23
  - distribution 23
  - organization 24
  - source code 23
- sscpp class library 1
  - features 2
- support, contacting xi

## T

- technical support xi
- TipcMon
  - monitoring RTclient data 7
  - monitoring SmartSockets data 7
- TipcMsg
  - accessing data fields 14
  - appending data fields 12
- TipcSrv
  - monitoring RTclient data 7
  - monitoring SmartSockets data 7
- SubjectSubscribe 14
- subscribing to a subject 14

## U

- utility functions 4
  - wrappers 4

## W

- wrappers 2