TIBCO SmartSockets™

.NET User's Guide and Tutorial

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.

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

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

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

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

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

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

Contents

Figures

Preface

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

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

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

This guide and tutorial is intended for software developers and project managers who want to familiarize themselves with the SmartSockets .NET API. Here you will find information about using the SmartSockets .NET API with Microsoft Visual Studio .NET, tutorials designed to get you started with SmartSockets .NET API, and a complete reference for the SmartSockets s.

Topics

About This Book

This User's Guide and Tutorial provides the detailed information you need to use and develop distributed applications with the TIBCO SmartSockets™ .NET API. This guide also includes a tutorial to help you quickly learn to use the SmartSockets .NET API. Before starting the tutorial, install SmartSockets. 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 .NET and C application program interfaces (APIs). This guide gives a brief overview of SmartSockets, emphasizing the differences between the .NET and C APIs.

For detailed reference information about the SmartSockets .NET classes, see the online reference information provided in MSDN Help 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 .NET API 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:

- basic knowledge of the Windows operating system
- working knowledge of the Microsoft .NET Framework
- understanding of general messaging and publish/subscribe concepts and terminology
- understanding of the SmartSockets messaging and publish/subscribe concepts described in the *TIBCO SmartSockets User's Guide*

Related Documentation

This guide and tutorial supplements the information presented in the following documents:

- *TIBCO SmartSockets Installation Guide*
- *TIBCO SmartSockets Tutorial*
- *TIBCO SmartSockets User's Guide*

Familiarize yourself with the information in these more general documents before working with this guide and tutorial, which is specific to the SmartSockets .NET API.

The following documents are also included in 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 Java User's Guide and Tutorial*
- *TIBCO SmartSockets .NET User's Guide and Tutorial*
- *TIBCO SmartSockets Utilities*
- *TIBCO SmartSockets C++, Java, and .NET Class Libraries* (These 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.

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 Using the TIBCO SmartSockets Class Library with Visual Studio .NET

This chapter describes how to use the TIBCO SmartSockets .NET assembly with Microsoft Visual Studio and how to register the assembly with the Global Access Cache for future sessions.

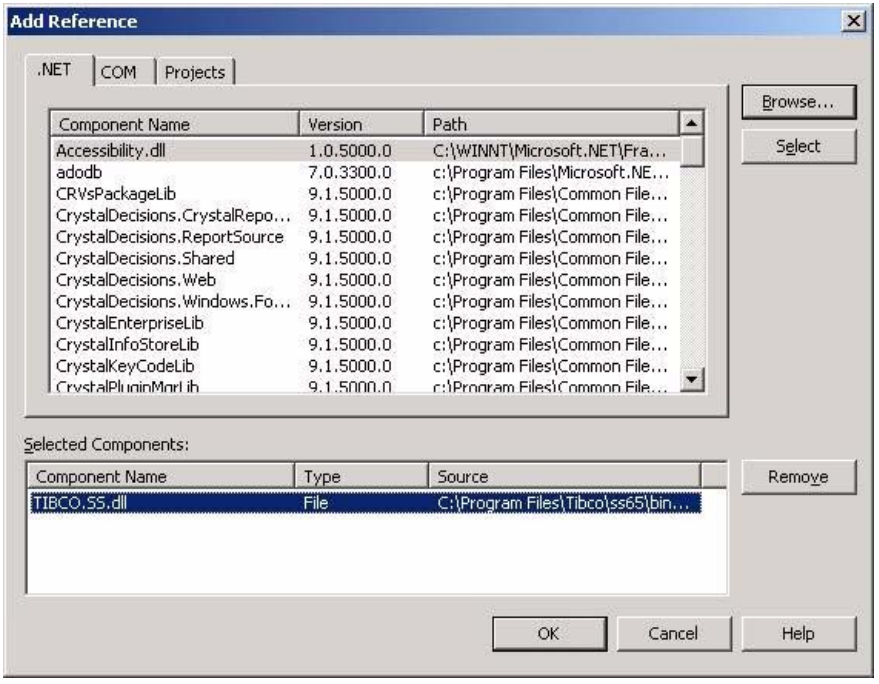
Topics

Referencing the TIBCO SmartSockets .NET Assembly

To access the .NET API from within Microsoft Visual Studio .NET, you need to add a reference to the .NET assembly. To add a reference to the TIBCO SmartSockets Assembly in your Visual Studio .NET project, perform these steps:

1. Select **Project**>**Add Reference**. The **Add Reference** dialog box appears, as shown in Figure 1.

Figure 1 The Add Reference Window



2. Select the TIBCO.SS.dll file as follows:

   a. Select the .NET tab

   b. Click **Browse** then navigate to `%RTHOME%\bin\i86_w32`, where `%RTHOME%` is the directory in which SmartSockets is installed.

   c. Double click **TIBCO.SS.dll**.

   The **TIBCO.SS.dll** file appears in the **Selected Components** list box, as shown in Figure 1.

3. Click **OK**.

You are now ready to use the TIBCO SmartSockets assembly in your project.

Using the Global Assembly Cache (GAC) Utility

If you want to make the SmartSockets .NET assembly available to all applications on the computer, you can install (register) the assembly in the GAC. This section provides procedures to both install and uninstall the assembly.

Installing the SmartSockets Assembly in the GAC

To install the .NET assembly with the GAC, perform these steps:

1. Verify that the GAC utility program is installed on the computer. This executable file, `gacutil.exe`, is usually stored in one of these locations:

   ```
   C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\bin\
   ```

   or

   ```
   C:\WINNT\Microsoft.NET\Framework\<version>\gacutil.exe
   ```

2. Run the GAC utility with the `/i` (install) option in a SmartSockets console window:

   gacutil_dir`\gacutil.exe /i "`*ss_lib_dir*`\TIBCO.SS.dll"`

   where *gacutil_dir* is the `full path to gacutil.exe` and where *ss_lib_dir* is the location of the SmartSockets library.

   Example:

   ```
   C:\WINNT\Microsoft.NET\Framework\v1.0.3705\gacutil.exe /i
   "%RTHOME%\bin\i86_w32\TIBCO.SS.dll"
   ```

Uninstalling the SmartSockets Assembly from the GAC

To uninstall the assembly from the GAC:

1. Verify that the GAC utility program is installed on the computer. (See step 1, above.)

2. Run the GAC utility with the `/u` (uninstall) option in a SmartSockets console window:

   gacutil_dir `/u "`*ss_lib_dir*`\TIBCO.SS.dll"`

   where *gacutil_dir* is the `full path to gacutil.exe` and where *ss_lib_dir* is the location of the SmartSockets library.

   Example:

   ```
   C:\WINNT\Microsoft.NET\Framework\v1.0.3705\gacutil.exe /u
   "%RTHOME%\bin\i86_w32\TIBCO.SS.dll"
   ```

Chapter 2 Getting Started with TIBCO SmartSockets .NET API

This chapter provides tutorial-style examples that you can work through to familiarize yourself with TIBCO SmartSockets and the TIBCO SmartSockets .NET API.

Topics

TIBCO SmartSockets .NET Configuration

You can configure TIBCO SmartSockets .NET in one of three ways:

- Using the `App.config` file from within your .NET solution
- Using an external file
- Using both the `App.config` file and an external file

In all cases, use the `Tut.loadOptions()` method to load the configuration file or files.

The naming standards for SmartSockets configuration options are consistent with those for SmartSockets java properties.

Use standard key, value attributes within the configuration options parameters.

The following table provides a quick reference for configuring SmartSockets .NET.

Table 1 Configuration File Quick Reference

Config File	Element Tag	Parameter Tag	Tut.loadOptions(); Parameters
App.config	`<appSettings>`	`<add>`	None
External File	`<SmartSockets>`	`<option name>`	URL or directory-path location of config file

Using the App.config File

If you are using the .NET Framework version 1.1 or later, you can configure SmartSockets options within the `app.config` file using the `<appSettings>` element with the `<add>` parameter. Here is an example:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
   <appSettings>
      <add key="ss.server_names" value="tcp:_node:5555" />
      <add key="ss.unique_subject" value="/CSharpTest" />
      <add key="ss.monitor_ident" value="CSharp Client" />
   </appSettings>
</configuration>
```

Use `Tut.loadOptions()` without parameters to load SmartSockets options from `App.config`.

Using an External File

You can configure SmartSockets options within an external XML file using the `<SmartSockets>` element with the `<option name>` node. Use standard key, value attributes with the `<option name>` element. Use `Tut.loadOptions(<string>)`, passing either a URL or a directory path to load the configuration file.

In this example, a URL identifies the location of the configuration file.

```
Tut.loadOptions("http://localhost/config/config.xml");
```

In this example, a directory path identifies the location of the configuration file.

```
Tut.loadOptions(""c:\\config\\config.xml"");
```

Below is an example configuration file.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<AnyRootNode>
   <SmartSockets>
      <option name = "ss.server_names" value = "tcp:_node:5555"/>
      <option name = "ss.unique_subject" value = "/xml_test_client"/>
      <option name = "ss.monitor_scope" value = "/..."/>
   </SmartSockets>
</AnyRootNode>
```

You can place the SmartSockets element at any level except at the root. If multiple SmartSockets elements are found, they are parsed in the order found, overriding previous options. `Tut.loadOptions()` will ignore any other elements, allowing for an easy addition of SmartSockets configuration attributes into existing XML files.

Using an External File with App.config

The options in App.config can act as a default configuration. You can then override individual options using an external configuration file. The last options loaded override any previously read options. Here is an example:

```
Tut.loadOptions();
Tut.loadOptions(""c:\\config\\config.xml"");
```

The first call to `Tut.loadOptions()` loads the configuration options from `App.config`. The second call loads the options from `config.xml`, overriding any options that were set inside `App.config` and exist in `config.xml`. Options that are set in `App.config`, but not in `config.xml` still retain their values.

Writing a SmartSockets Program in Visual Studio .NET

This section provides a procedure to write a simple C# SmartSockets program in Visual Studio .NET and descriptions of the classes used in the example.

The first part of the SmartSockets program invokes a connection to an RTserver and sets up the delegates that will process messages and errors from the server. The second part sends and receives a message to and from a subscribing RTclient.

Introduction to the Classes, Events, and Delegates used in this Program

This section briefly describes the classes used in the example program, which follows. For detailed information about all SmartSockets Classes, Events, and Delegates for .NET, see *TIBCO SmartSockets .NET API Reference* (HTML only).

TipcMt

The TipcMt methods create and retrieve information about message types, which are templates for messages. Many of the constants associated with TipcMt can be found in TipcMt fields; for example, the message type number for an “info” message would be TipcMt_Fields.INFO.

TipcMsg

The TipcMsg methods construct, manipulate and destroy messages as well as constructing and accessing the fields in the data section of a message.

TipcSvc

TipcSvc is a static factory class for creating instances of the TipcMt, TipcMsg, TipcConnClient, TipcConnServer, and TipcSrv classes. Objects of the required concrete classes that are created with this factory class.

```
TipcMsg msgOut = TipcSvc.createMsg(TipcMt_Fields.INFO)
```

TipcMsgEvent

TipcMsgEvent occurs when a message is processed, as in TipcSrv.mainloop.

Connection process events are raised while processing a message. This callback type is the most frequently used. A process delegate method is called for every of message received. When any message of any type is processed by calling process or mainLoop, the process callback is called.

TibcMsgHandler Delegate

Represents the method that will handle the TipcMsgEvent event.

```
Srv.TipcMsgEvent += new TipcMsgHandler(this.msgHandler);
```

TipcMsgEventArgs

This class contains event data for the TipcMsgEvent.

TipcSrv

TipcSrv methods communicate with RTserver to receive and process messages, for example, TipcSrv.mainLoop

TipcException

Superclass of all SmartSockets exceptions. This class defines an error number which may be set for some errors.

```
catch (TipcException ex) {
}
```

TipcDefs

The TipcDefs structure contains constants used across all classes.

Example: TipcDefs.CONN_FULL

TipcProcessCb

The user interface for connection process callbacks.

Connection process callbacks are executed while processing a message. This callback type is the most frequently used. A process callback can be called for a specific type of message, on a specific subject (destination), or created globally and called for all messages. For example, a process callback can be created for the INFO message type. When any message of that type is processed by calling process or mainLoop, the process callback is called. If the process callback is created globally, it is called for all INFO messages as well as any other type of message.

TipcProcessCB is included in this section for completeness only. TIBCO recommends that you use events and event handlers instead.

Writing the Program

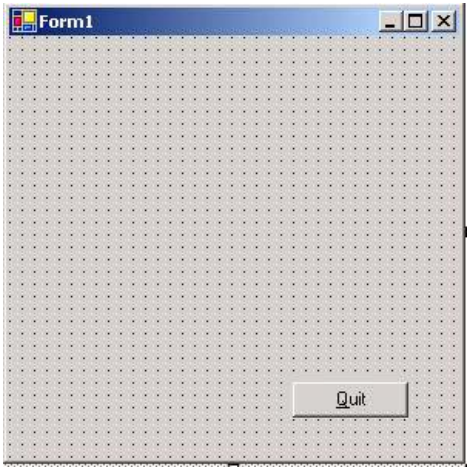
Before you begin, start a new Visual C# Windows Application project and add the TIBCO.SS reference. (See Referencing the TIBCO SmartSockets .NET Assembly on page 2)

Perform these steps to create the RTserver connection program:

Task A Create a Quit Button

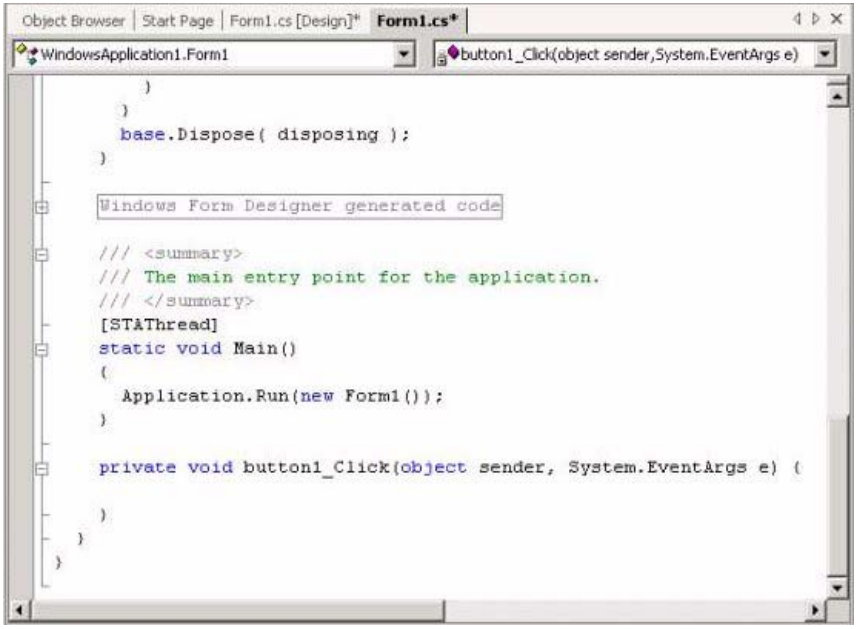
1. From the Windows Form Controls, select the Button icon and place a button on the Form, as pictured in Figure 2.

Figure 2 Form Window



2. In the Properties window, to the right of the text field, type: **`Quit`**. The button changes in real-time, reflecting what you type—the button should now have the label **Quit**.
3. Double-click the Quit button you just created. The Code window appears, as shown in Figure 3.

Figure 3 Code Window



4. At the current insertion point in the code window, type:

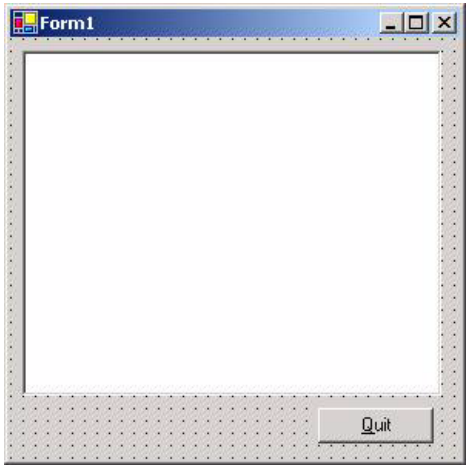
```
private bool running = false;
Application.Exit();,
```

   Indent just as you would when writing other programs.

5. Return to the Form window.
6. Test the **Quit** button:
   a. On the toolbar, click **Start**. A window appears displaying the **Quit** button.
   b. Click **Quit** to close Form1 and return to the code window.
7. Add a text box to the form, defining it as follows:
   — Set the Multiline Property to true.
   — Delete the Text property contents.
   — Size the window appropriately.

   The resulting window should be similar to the one in Figure 4.

Figure 4 Form1 with Text Box



Task B Create a Message Handler

1. Scroll up to the top of the code window and add the "using TIBCO.SMARTSOCKETS;" directive:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using TIBCO.SMARTSOCKETS;
```

2. Add the following fields to the Form1 class:

```
public TipcSrv Srv;
private bool running = false;
```

3. Open the **Method Wizard** by right-clicking on the **Form1** class in the project explorer, then select **Add->Add Method**.

4. Create a message handler method with a the following characteristics:

   — **Method access**: `public`

   — **Return type**: `void`

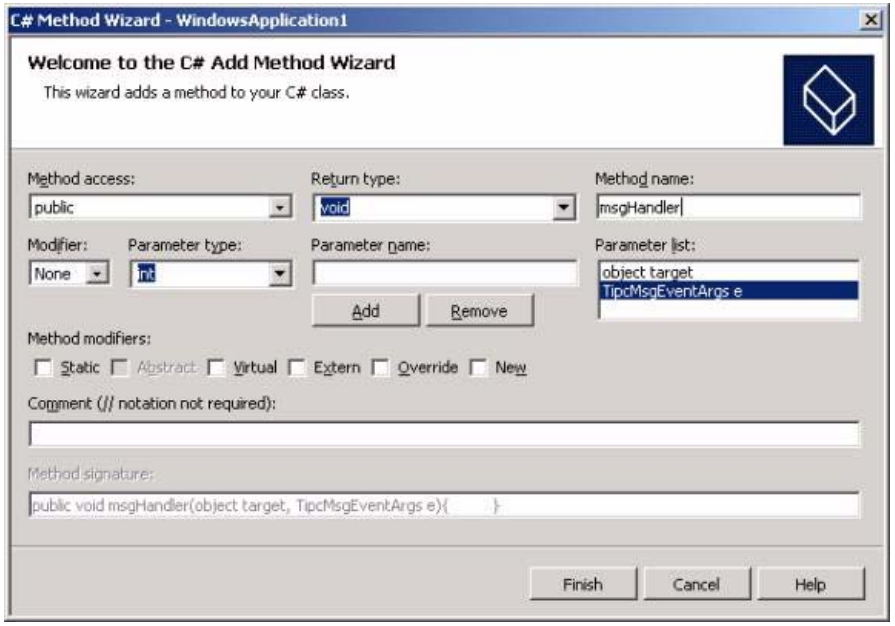
   — **Method name**: `msgHandler`

   — **Parameter name**: `object target`

   — **Parameter name**: `TipcMsgEventArgs e`

The **C# Method Wizard** window should look similar to the one shown in Figure 5.

Figure 5 C# Method Wizard



5. Add code to `msgHandler` such that the method looks like this:

```
public void msgHandler(object target, TipcMsgEventArgs e) {

   try {
      TipcMsg msgOut = TipcSvc.createMsg(TipcMt_Fields.INFO);
      msgOut.Dest = e.Msg.Sender;
      msgOut.appendStr("Message Received!");
      Srv.send(msgOut);
      Srv.flush();
   }

   catch (TipcException ex) {
      MessageBox.Show(ex.Message, "msgHandler:  Exception",
         MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
   }

}
```

Task C Create Another Message Handler

6. Create a second message handler, identical to the one you created in Task B, except name it **msgHandler_Hail**.

7. Add code to `msgHandler_Hail` such that the method looks like this:

```
public void msgHandler_Hail(object target, TipcMsgEventArgs e) {

   try {
      if (e.Msg.Type.Num == TipcMt_Fields.INFO &&
         e.Msg.Dest.CompareTo("/hail") == 0) {
         this.textBox1.Text += e.Msg.nextStr() +
                               System.Environment.NewLine;
      } // if
   }
   catch (Exception ex) {
      MessageBox.Show(ex.Message, "msgHandlerHail:  Exception",
         MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
   }

   }
```

8. Add code to the form's constructor, so that the constructor looks like this:

```
public Form1()
{
   //
   // Required for Windows Form Designer support
   //
   InitializeComponent();

    try {
     Srv = TipcSvc.Srv;
     Srv.setOption("ss.unique_subject", "/cs_example");
     Srv.setOption("ss.server_names", "tcp:_node");

     /* add the delegates */
     Srv.TipcMsgEvent += new TipcMsgHandler(this.msgHandler);
     Srv.TipcMsgEvent += new TipcMsgHandler(this.msgHandler_Hail);

     Srv.create(TipcDefs.CONN_FULL);
     Srv.setSubjectSubscribe("/hail", true);
    }
    catch (Exception ex) {
     MessageBox.Show(ex.Message, "msgHandlerHail:  Exception",
       MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
         Application.Exit();
    }
}
```

9. Add code to the Form's Activated event such that the event looks like this:

```
private void Form1_Activated(object sender, System.EventArgs e) {
   while (running) {
      Srv.mainLoop(0.0);
      Application.DoEvents();
   }
}
```

Note, the recommended method of handling the SmartSockets event loop (TipcSrv.mainLoop) is through another thread. The Form1_Activated event was used only for simplicity and brevity.

Task D Test the Program

1. Verify that RTserver is running on the local machine.
2. Launch Form1 by selecting **Debug->Launch without Debugging** from the menu bar, or enter CTRL+F5.
3. Start RTmonitor with the `-runtime` option from the SmartSockets command prompt.
4. Enter the following commands:

```
MON> connect

09:01:10: TAL-SS-00088-I Connecting to project <rtworks> on
<tcp:_node:5555> RTserver
09:01:10: TAL-SS-00089-I Using tcp protocol
09:01:10: TAL-SS-00091-I Message from RTserver: Connection
established.
09:01:10: TAL-SS-00096-I Start subscribing to subject
</_CLSLAP01_2108>
09:01:10: TAL-SS-00111-I Start subscribing to subject </_CLSLAP01>
09:01:10: TAL-SS-00111-I Start subscribing to subject </_all>
09:01:10: TAL-SS-00111-I Start subscribing to subject </_mon>

MON> send info /hail "Hello!"

Sent info message to /hail subject.

MON> run 1 1

Received an unexpected message.
```

The message below is what was constructed and sent back to RTmonitor in the msgHandler delegate. The messageHandler_Hail delegate displays the message contents in Form1's text box.

```
type = info
sender = </cs_example>
sending server = </_CLSLAP01_2008>
dest = </_CLSLAP01_2108>
app = <rtworks>
max = 2048
size = 40
current = 0
read_only = false
priority = 0
compression = false
delivery_mode = best_effort
ref_count = 1
seq_num = 0
resend_mode = false
user_prop = 0
arrival_timestamp = 09:02:10
data (num_fields = 1):
  str "Message Received!"
Processed a info message.
```

```
MON> send info /hail "Hello Again!"
```

5. Form1 displays both messages:

Example Programs

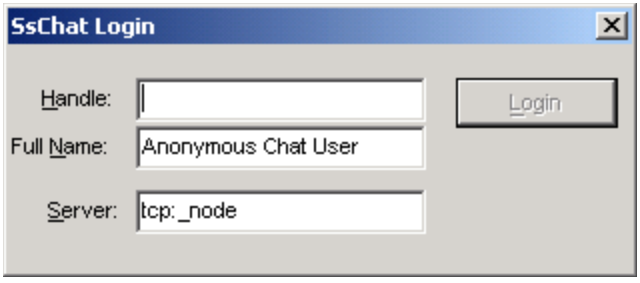
Now that you have written a simple TIBCO SmartSockets program, this section presents two programs that use some of the more advanced SmartSockets features. These programs are not presented in a tutorial format; instead portions of the Visual Studio .NET Visual Basic 7 code for each program are reviewed and key elements are discussed.

SSChat: Multi-User Chat Room

The SSChat program is an RTclient that implements a simple real-time, multi-user chat room. The example files are located in the installation directory under `Examples\Microsoft.NET\vb7\sschat`. The SSChat program is written using the SmartSockets publish-subscribe technology to implement this function with a minimum amount of code. The entire SSChat program is under 200 lines of Visual Basic code.

The SSChat application starts with the Login window, as shown in Figure 9. The Login window is a standard Visual Studio .NET form named `frmLogin`. It allows the user to input their full name and handle, and to specify an RTserver to connect to.

Figure 6 Figure 9 SSChat Login Window



The essential details of SSChat's implementation are included within the main form, `frmMain`.

Exploring frmMain

There are several form class variables defined as follows:

```
Dim mtHi As TipcMt
Dim mtBye As TipcMt
Dim msgHi As TipcMsg
Dim msgOut As TipcMsg
Dim RTserver_Conn As TipcSrv
Dim HiCb_parms As New Collection

Dim tsThread As ThreadStart
Dim trtsThread As Thread
Dim rtthrdclass As rtthrdclass
```

The first two variables, `mtHi` and `mtBye`, hold TipcMt objects specifying two of the custom message types SSChat defines. The second two, msgHi and msgOut, hold pre-built messages that are sent multiple times during an execution of SSChat. Finally, RTserver_Conn is a TipcSrv object that manages the connection to RTserver.

Looking at the `mainInit()` method shows the actions taken upon entering the chat room. The custom message types are registered and the server name is configured. A connection to RTserver is created, process callbacks are instantiated using user callbacks implemented from TipcProcessCb, and a “Hi” (mtHi) message is constructed containing the user's name and chat handle, then sent. Please note that in this example, callbacks are used for completeness. The use of delegates is recommended over callbacks, and is demonstrated in the next example, WhoWhere.

```
Public Sub mainInit()
   Dim btn As Short
   Dim server_names As String
   Dim hi_cb As New HiCb
   Dim bye_cb As New ByeCb
   Dim data_cb As New DataCb

   ctlUser.Text = fmLogin.DefInstance.ctlHandle.Text
   ctlName.Text = fmLogin.DefInstance.ctlName.Text
   ctlScript.ForeColor = System.Drawing.Color.Blue
   ctlScript.Text = "[ " & Format(Now(), "General Date") & " ]"

   If mtHi Is Nothing Then
      mtHi = TipcSvc.createMt("hi", 1, "str str int2")
      mtBye = TipcSvc.createMt("bye", 2, "str")
      msgOut = TipcSvc.createMsg(TipcSvc.lookupMt("string_data"))
   End If

   msgOut.Dest = "_all"
   msgOut.appendStr(fmLogin.DefInstance.ctlHandle.Text)
   msgHi = msgOut.Clone
   msgHi.Type = mtHi
   msgHi.appendStr(fmLogin.DefInstance.ctlName.Text)
   msgHi.appendInt2(Int(CDbl(True)))
```

```
    server_names = ""
    If Len(fmLogin.DefInstance.ctlServer.Text) > 0
       Then server_names = fmLogin.DefInstance.ctlServer.Text
    End If

    RTserver_Conn = TipcSvc.Srv
    RTserver_Conn.setOption("ss.unique_subject",
       fmLogin.DefInstance.ctlHandle.Text)
    RTserver_Conn.setOption("ss.project", "rtworks")
    RTserver_Conn.setOption("ss.server_names", server_names)

    RTserver_Conn.create()

    HiCb_parms.Add(RTserver_Conn)
    HiCb_parms.Add(msgHi)
    HiCb_parms.Add(msgOut)
    HiCb_parms.Add(ctlScript)

    RTserver_Conn.addProcessCb(bye_cb, mtBye, ctlScript)
    RTserver_Conn.addProcessCb(data_cb,
       TipcSvc.lookupMt("string_data"), ctlScript)
    RTserver_Conn.addProcessCb(hi_cb, mtHi, HiCb_parms)

    RTserver_Conn.setSubjectSubscribe("_all", True)

    RTserver_Conn.send(msgHi)
    RTserver_Conn.Flush()

    '
    ' Here, we need a thread to allow us to do the MainLoop to
    ' receive messages from the server. So, create the appropriate
    ' objects, and set the server conn to
    ' use within the thread. Then, start the thread.

rtthrdclass = New rtThrdClass
tsThread = New ThreadStart(AddressOf rtthrdclass.mainThreadProc)
trtsThread = New Thread(tsThread)
rtthrdclass.thrdServerConn = RTserver_Conn
rtthrdclass.thrdStop = False
trtsThread.Start()
End Sub
```

By cloning the `msgOut` message, the `Dest` property is the same for `msgHi`. Appending `True` requests other SSChat clients to reply upon receipt of the message.

To receive messages, the SSChat application must maintain a message loop. This is done in a separate thread. It is a simple loop that processes events and checks for messages without a timeout to provide the best possible GUI response.

```
Public Class rtThrdClass
   Public thrdServerConn As TipcSrv
   Public thrdStop As Boolean
   Public Sub mainThreadProc()
        While thrdStop = False
            thrdServerConn.MainLoop(0)
            Thread.Sleep(0)
        End While
   End Sub
End Class
```

The loop is terminated when the `thrdStop` flag is set to `true` in `mainCleanup()`.

Once the form has been loaded, SSChat is in its primary mode, waiting for keystrokes from the user or messages from RTserver. Examine what happens when a key is pressed. Note that when you press the Enter key, the current text is sent out to the other chat participants.

```
   Private Sub ctlMessage_KeyDown(…) Handles ctlMessage.KeyDown
      Dim KeyCode As Short = eventArgs.KeyCode
      Dim Shift As Short = eventArgs.KeyData \ &H10000

      Dim str_Renamed As String

      If (Shift And VB6.ShiftConstants.ShiftMask) = 0 Then
         If KeyCode = System.Windows.Forms.Keys.Return Then
            msgOut.NumFields = 1
            str_Renamed = ctlMessage.Text
               If VB.Left(str_Renamed, 1) = Chr(13) Then
                  str_Renamed = VB.Right(str_Renamed,
                     Len(str_Renamed) - 2)
               End If

               msgOut.appendStr(str_Renamed)
               RTserver_Conn.send(msgOut)
               RTserver_Conn.Flush()
               ctlMessage.Text = ""
            End If
         End If
      End Sub
```

Notes:

- Setting the `NumFields` property to `1` eliminates all but the first field of the message.
- `msgOut` is already mostly constructed; all you have to do is add the chat text, send and flush to guarantee immediate delivery. Using a previously constructed message will improve performance.

Now that you have reviewed the sending process of chat text from SSChat, take a look at the three message processing events that handle the receiving process. Note that these events are implementing the TipcProcessCb interface. The TipcMsg event from the first `ctlHiCb` receives the "Hi" messages:

```
Friend Class HiCb
   Implements TipcProcessCb

   Public Sub process(ByVal msg As TIBCO.SMARTSOCKETS.TipcMsg,
                      ByVal arg As Object) Implements
                        TIBCO.SMARTSOCKETS.TipcProcessCb.process
      Dim msgIn As TipcMsg
      Dim msgHi As TipcMsg
      Dim msgOut As TipcMsg
      Dim srv_conn As TipcSrv
      Dim cb_parms As Collection
      Dim ctlScript As TextBox
      Dim inHandle As String
      Dim inUserName As String
      Dim inInt As Int32

      msgIn = msg
      cb_parms = arg

      msgIn.Current = 0
      inHandle = msgIn.nextStr()
      inUserName = msgIn.nextStr()
      inInt = msgIn.nextInt2()
      srv_conn = cb_parms.Item(1)
      msgHi = cb_parms.Item(2)
      msgHi.Current = 0
      msgOut = cb_parms.Item(3)
      msgOut.Current = 0
      ctlScript = cb_parms.Item(4)

      If inUserName.CompareTo(msgOut.nextStr()) = 0 Then
         If inInt Then
            msgHi.Dest = msgIn.Sender
            msgHi.NumFields = msgHi.NumFields - 1
            msgHi.appendInt2((Int(CDbl(False))))
            srv_conn.send(msgHi)
            srv_conn.flush()
         End If

      End If
```

```
      ctlScript.AppendText(ControlChars.NewLine &
         ControlChars.NewLine)
      ctlScript.ForeColor = System.Drawing.Color.Blue

      ctlScript.AppendText("[ " + inHandle + " connected as " +
         inUserName + " ]")

   End Sub
End Class
```

Now look at the code that handles the “Bye” messages, sent when a user leaves the chat room, terminating the SSChat process. It is very similar to the “Hi” message event handler, except it does not send any replies.

```
Friend Class ByeCb
Implements TipcProcessCb
   Public Sub process(ByVal msg As TIBCO.SMARTSOCKETS.TipcMsg,
                      ByVal arg As Object) Implements
                        TIBCO.SMARTSOCKETS.TipcProcessCb.process
      Dim msgIn As TipcMsg
      Dim ctlScript As TextBox

      msgIn = msg
      ctlScript = arg

      ctlScript.AppendText(ControlChars.NewLine &
         ControlChars.NewLine)
      ctlScript.ForeColor = System.Drawing.Color.Blue
      ctlScript.AppendText("[ " + msgIn.nextStr + "
         disconnected ]")

   End Sub

End Class
```

The event handler for chat data messages is shown below (similar to the other two message events, it updates the output window with the originating chat user's handle and chat text):

```
Friend Class DataCb
   Implements TipcProcessCb

   Public Sub process(ByVal msg As TIBCO.SMARTSOCKETS.TipcMsg,
                      ByVal arg As Object) Implements
                        TIBCO.SMARTSOCKETS.TipcProcessCb.process

      Dim msgIn As TipcMsg
      Dim ctlScript As TextBox

      msgIn = msg
      ctlScript = arg

      ctlScript.AppendText(ControlChars.NewLine &
                           ControlChars.NewLine)
      ctlScript.AppendText(msgIn.nextStr() + ": ")
      ctlScript.AppendText(msgIn.nextStr())

   End Sub
End Class
```

In the code above, the first `msgIn.nextStr()` is the chat handle and the second `msgIn.nextStr()` retrieves the actual user message from TipcMsg.

Finally, SSChat's mainCleanup()subroutine, which stops the listener thread, sends the “Bye” message, and disconnects from RTserver (if a connection has been established) is shown below:

```
Public Sub mainCleanup()
   Dim msgBye As TipcMsg
   If Not trtsThread Is Nothing Then
      '
      ' Tell the thread to stop
      rtthrdclass.thrdStop = True
      '
      ' Wait until the thread has ended
      trtsThread.Join(10000)
      trtsThread = Nothing
   End If
   If Not RTserver_Conn Is Nothing Then
      If RTserver_Conn.ConnStatusEx <> TipcDefs.CONN_NONE Then
         msgBye = TipcSvc.createMsg(mtBye)
         msgBye.Dest = msgOut.Dest
         msgOut.Current = 0
         msgBye.appendStr(msgOut.nextStr())
         RTserver_Conn.send(msgBye)
         RTserver_Conn.flush()
         RTserver_Conn.destroy(TipcDefs.CONN_NONE)
      End If
      RTserver_Conn = Nothing
   End If
```

```
   If Not mtBye Is Nothing Then
      mtBye.destroy()
      mtBye = Nothing
   End If
   If Not mtHi Is Nothing Then
      mtHi.destroy()
      mtHi = Nothing
   End If
End Sub
```

Data Flow

The data flow in the SSChat program is:

1. Chat users joining the room are announced to the group members by publishing a “Hi” message containing information about the new user.
2. Other SSChat RTclients receive the new user's announcement and reply directly and exclusively to the originating RTclient, by publishing a “Hi” message back.
3. Upon receiving a “Hi” or “Bye” message, an SSChat RTclient updates its output window with the status change indicating which other client entered or left the room.
4. As data messages are received, SSChat displays these messages to its chat window.

WhoWhere: Electronic Message Board

The WhoWhere program is a graphical RTclient, implementing an in/out message board useful for tracking employee whereabouts. The WhoWhere program is located in the installation directory under `Examples\Microsoft.NET\vb7\whowhere`.

WhoWhere is an electronic counterpart to the sign in and out boards commonly used in corporate offices. WhoWhere uses TIBCO SmartSockets publish-subscribe technology to update other users message boards. Other SmartSockets features demonstrated by this program are:

- Custom message types
- Process and error Delegates
- Messages within messages
- Hierarchical naming scheme for publish-subscribe messages
- Publishing to subjects with multiple subscribers as well as individual RTclients

WhoWhere tracks employee whereabouts. Each employee belongs to a department. The department an employee belongs to specifies the SmartSockets subject used to set apart message board groups. This allows potentially thousands of users in hundreds of departments to be present on the same LAN or WAN, without any conflict.

The WhoWhere application is composed of several Visual Studio .NET forms and one code module. Most of the forms (those for logging in, managing the configuration and specifying leave information) are handled with the usual Visual Studio .NET techniques; examination of the source code should be fairly self-explanatory. The important parts of the application are handled by the Display Board form and the `modGlobals` module.

First, look at some of the data structures used in the modGlobals module:

```
Public Structure configType
   Dim Name As String
   Dim Password As String
   Dim Lunch As Short
   Dim Email As String
   Dim Homepage As String
   Dim MailApp As String
   Dim WebApp As String
   Dim server As String
   Dim Alerts As Short
   Dim Department As String
   Dim EmailBrowser As Short
End Structure

Public Config As configType
```

The configType structure holds the local configuration information. This data is stored in the Windows registry by the SaveSettings subroutine and reloaded with LoadSettings. This allows the configuration to be persistent between executions of the application. The data includes the user's name, password and other configuration information. A subset of this information is maintained for all the other known employees on the message board as well, as shown in the userType data structure:

```
   Public Structure userType
      Dim Who As String
      Dim Email As String
      Dim Homepage As String
      Dim Message As String
      Dim Where As String
      Dim ReturnInfo As String
   End Structure
```

```
Public Myself As userType
Public Users(maxUsers) As userType
Public nUsers As Integer

Public wwSubject As String
Public Srv As TipcSrv
Public mtAnnounce As TipcMt
Public mtUpdate As TipcMt
Public mtResponse As TipcMt

Public Enum wwMessageTypes
   wwAnnounce = 100
   wwUpdate
   wwResponse
End Enum

Public AnnounceCbParms As New Collection
```

`Myself` holds a copy of this user's message board information; `Users()` is the array that holds the actual message board information. `nUsers` is the number of users in the array and therefore the number of users displayed on the board. `wwSubject` holds the publish-subscribe subject name used for the WhoWhere client communication.

A global handle to the RTserver connection in the application is held in the Srv variable. Additionally, WhoWhere defines three new message types identified by the numbers `100`, `101`, and `102` as the enumeration `wwMessageTypes` specifies; `mtAnnounce`, `mtUpdate` and `mtResponse` act as global references to these message type objects that are created. Next, the `initializeClient` subroutine connects to RTserver and assigns user-defined delegates to SmartSockets events.

```
Public Sub initializeClient()

   displayErrors = True

   If Srv Is Nothing Then
      ' get a handle to our server connection
      Srv = TipcSvc.Srv

      ' set our options before we connect
      Srv.setOption("ss.server_names", Config.server)
      Srv.setOption("ss.project", programName)
      Srv.create(TipcDefs.CONN_FULL)

      destroyMts()

      ' create custom message types
      mtAnnounce = TipcSvc.createMt("wwAnnounce",
                     wwMessageTypes.wwAnnounce, "int2 msg")
      mtResponse = TipcSvc.createMt("wwResponse",
                     wwMessageTypes.wwResponse, "int2 msg")
```

```
        ' update is the msg that gets sent inside announce
        ' and response, and also by itself for updating status
        mtUpdate = TipcSvc.createMt("wwUpdate",
        wwMessageTypes.wwUpdate, "int2 str str str str str")

        ' install event handlers for message types
        ' in this example, we use delegates instead of
        ' callbacks.  For .NET applications, delegates
        ' are much more efficient.
        AddHandler Srv.TipcMsgEvent, AddressOf MsgHandler
        AddHandler Srv.TipcErrorEvent, AddressOf ErrorHandler

    End If

    wwSubject = "/" & programName & "/" & Config.Department
    Dim status As Boolean
    status = Srv.getSubjectSubscribe(wwSubject)
    If status = False Then
       Srv.setSubjectSubscribe(wwSubject, True)
    End If
    Srv.flush()

    ' send a message announcing our arrival
    doUpdate()

End Sub
```

Only message board traffic published to the `wwSubject` subject is seen by this client. It is this hierarchical naming feature of SmartSockets that accounts for its scalability. Without changing the client, and with only one level of partitioning (the department), a large number of separate message boards can coexist. This is demonstrated in the `RTserver.SubjectSetSubscribe` statement in the `initalizeClient()` subroutine above. As illustrated in the code, a call to the doUpdate subroutine is made. This sends the equivalent of a “Hello” message from this RTclient to the other users displaying this department's message board. The following code for doUpdate builds and sends an announcement message using the mtAnnounce object as a reference (notice how a second SmartSockets message of type mtUpdate is included inside the announcement message):

```
Public Sub doUpdate()

    ' build and send an anouncement message
    ' for initially joining a message board subject
    ' or doing an 'update all'
    Dim am, myData As TipcMsg
    myData = TipcSvc.createMsg(mtUpdate)
```

```
    With myData
       .appendInt2(messageFormat)
       .appendStr(Myself.Who)
       .appendStr(Myself.Email)
       .appendStr(Myself.Homepage)
       .appendStr(Myself.Where)
       .appendStr(Myself.Message)
       .appendStr(Myself.ReturnInfo)
    End With

    am = TipcSvc.createMsg(mtAnnounce)
    am.appendInt2(messageFormat)
    am.appendMsg(myData)
    am.Dest = wwSubject
    Srv.send(am)
    Srv.flush()

End Sub
```

The following subroutine, `publishMyStatus`, is used to send an update message when the current user's status changes, for example, when they leave or return to the office. A message of type `mtUpdate` is sent alone this time, not included inside another message.

```
Private Sub publishMyStatus()

    ' build and send just an update message when
    ' our status changes (go to lunch, return, etc.)
    Dim myData As TipcMsg

    myData = TipcSvc.createMsg(mtUpdate)
    With myData
       .appendInt2(messageFormat)
       .appendStr(Myself.Who)
       .appendStr(Myself.Email)
       .appendStr(Myself.Homepage)
       .appendStr(Myself.Where)
       .appendStr(Myself.Message)
       .appendStr(Myself.ReturnInfo)
       .Dest = wwSubject
    End With

    Srv.send(myData)
    Srv.flush()

End Sub
```

The `setAway` subroutine updates the buttons available on the WhoWhere graphical user interface (GUI). There are two subroutines, goAway and comeBack, that make calls to `publishMyStatus`, as shown here:

```
Public Sub goAway()
    publishMyStatus()
    frmBoard.DefInstance.setAway(True)
End Sub
```

```
   Public Sub comeBack()
      Myself.Where = inStatus
      publishMyStatus()
      frmBoard.DefInstance.setAway(False)
   End Sub
End Module
```

The Display Board Form

The Display Board Form is the main user-interface object of the WhoWhere application. This form displays the message board with dynamic updating and allows the user to interact with the program through the command buttons. In the next example, the form's `Load` subroutine calls the other initialization routines such as `initializeGUI`, which positions and sizes various screen elements. `initializeClient` configures the environment for SmartSockets, assigns delegates to SmartSockets events and manages subscriptions to the relevant subjects.

A thread is also created to run a loop processing messages in the background. This is how the application receives and processes messages using the TipcSrv.Mainloop method. Note that the loop must be terminated by setting a flag in the frmBoard_Close subroutine.

```
   Private Sub frmBoard_Load(ByVal eventSender As System.Object,
                             ByVal eventArgs As System.EventArgs)
            Handles MyBase.Load
      Me.Left = Val(GetSetting(regAppName, "WindowPos", "Left",
      Str(System.Windows.Forms.Screen.PrimaryScreen.Bounds.Width -
            Me.Width)))
      Me.Top = Val(GetSetting(regAppName, "WindowPos",
            "Top", "0"))
      Me.Text = programName & " v" & programversion
      initializeGUI()
      noErrBox = True
      initializeClient()

      '
      ' Here, we need a thread to allow us to do the MainLoop to
      ' receive messages from the server.  So,
            create the appropriate
      ' objects, and set the server connection to
      ' use within the thread.  Then, start the thread.

      rtthrdclass = New rtThrdClass()
      tsThread = New ThreadStart(
            AddressOf (rtthrdclass.mainThreadProc)
      trtsThread = New Thread(tsThread)
      rtthrdclass.thrdServerConn = Srv
      rtthrdclass.thrdStop = False
      trtsThread.Start()
```

```
      Me.Show()
      Me.Visible = True

   End Sub
```

Here is the rtThreadClass, demonstrating the use of TipcSrv.Mainloop to listen for and process messages.

```
Public Class rtThrdClass
   Public thrdServerConn As TipcSrv
   Public thrdStop As Boolean
   Public Sub mainThreadProc()
      While thrdStop = False
         thrdServerConn.MainLoop(0)
         Thread.Sleep(0)
      End While
   End Sub
End Class
```

The form's Unload code saves the program settings and disconnects from the RTserver.

```
   Private Sub frmBoard_Closed(ByVal eventSender As System.Object,
                               ByVal eventArgs As System.EventArgs)
                               Handles MyBase.Closed
      DoLoop = False
      SaveSetting(regAppName, "WindowPos", "Left",
          Me.Left.ToString)
      SaveSetting(regAppName, "WindowPos", "Top", Me.Top.ToString)
      saveSettings()
      If Not rtthrdclass Is Nothing Then
         '
         ' Tell the thread to stop
         rtthrdclass.thrdStop = True
         '
         ' Wait until the thread has ended
         trtsThread.Join(10000)
         trtsThread = Nothing

         Srv.destroy()
      End If

   End Sub
```

As shown in the next example, the comeBack and goAway subroutines are called when the absence or return (btnBack) buttons are clicked:

```
   Private Sub btnHome_Click(ByVal eventSender As System.Object,
                             ByVal eventArgs As System.EventArgs)
                             Handles btnHome.Click
      Myself.Where = "HOME"
      Myself.Message = "(left for the day)"
      Myself.ReturnInfo = "the next work day"
      goAway()
      btnBack.Focus()
   End Sub
```

```
   Private Sub btnLunch_Click(ByVal eventSender As System.Object,
                              ByVal eventArgs As System.EventArgs)
                              Handles btnLunch.Click
      Myself.Where = "LUNCH"
      Myself.Message = "(at lunch)"
      Myself.ReturnInfo = "at " &
         Format(DateAdd(Microsoft.VisualBasic.DateInterval.Minute,
                        Config.Lunch, TimeOfDay))
      goAway()
      btnBack.Focus()
   End Sub

   Private Sub btnExtended_Click(ByVal eventSender As
System.Object,
                                 ByVal eventArgs As
System.EventArgs)
                     Handles btnExtended.Click
        frmExtended.DefInstance.ShowDialog()
        If extendedCancel Then Exit Sub
        goAway()
        btnBack.Focus()
   End Sub

   Private Sub btnBack_Click(ByVal eventSender As System.Object,
                             ByVal eventArgs As System.EventArgs)
                     Handles btnBack.Click
        comeBack()
   End Sub
```

The displayBoard subroutine is the most important in terms of user interface; it re-populates the message board with data from the Users() array. The board has nUsers+1 rows; the extra is used to display the column headings.

```
   Public Sub displayBoard()
        Dim i As Short
        Dim wh As String

        If board.Items.Count > 0 Then
            board.Items.Clear()
        End If

        For i = 0 To nUsers - 1
            Dim itm As New ListViewItem(Users(i).Who, i)
            wh = Users(i).Where
            If wh <> inStatus Then
               wh = wh & " - returns " & Users(i).ReturnInfo
            End If
            itm.SubItems.Add(wh)
            board.Items.Insert(i, itm)
        Next i
        btnMail.Visible = False
        btnWeb.Visible = False
```

```
        If Config.Alerts Then
            Beep()
        End If

   End Sub
```

The `msgHandler` delegate handles announce, response, and update messages. It then passes the messages to various subroutines. The `newUser` subroutine adds a user to the message board. Note that is was registered in `initializeClient()`. As shown this example, if `msgHandler` processes an announcement message, it responds by publishing a response message directly back to the originator:

```
' This is the message handler delegate to handle messages when they
arrive.
 Public Sub MsgHandler(ByVal target As Object,
                       ByVal args As TipcMsgEventArgs)
   Dim msg As TipcMsg
   Dim mt As TipcMt

   msg = args.Msg
   mt = msg.Type

   If mt.Num = wwMessageTypes.wwAnnounce Then
      HandleAnnounceMessage(msg)
      Exit Sub
   End If
   If mt.Num = wwMessageTypes.wwResponse Then
      HandleResponseMsg(msg)
      Exit Sub
   End If
   If mt.Num = wwMessageTypes.wwUpdate Then
      HandleUpdateMsg(msg)
      Exit Sub
   End If
   MsgBox("Received unexpected message of type " & mt.Name)

End Sub
```

The following are called from the msgHandler delegate to handle different message types.

```
Public Sub HandleAnnounceMessage(ByVal msg As TipcMsg)
        Dim mver As Short
        Dim m2 As TipcMsg
        Dim resp As TipcMsg
        Dim myData As TipcMsg

        mver = msg.nextInt2
        If mver > messageFormat Then
            frmBoard.DefInstance.wrongMessageFormat("ANNOUNCE",
mver)
```

```
        Else
            m2 = msg.nextMsg
            mver = m2.nextInt2
            frmBoard.DefInstance.newUser(m2)
            resp = TipcSvc.createMsg(mtResponse)
            resp.Dest = msg.Sender
            resp.appendInt2(mver)

            myData = TipcSvc.createMsg(mtUpdate)
            With myData
               .appendInt2(messageFormat)
               .appendStr(Myself.Who)
               .appendStr(Myself.Email)
               .appendStr(Myself.Homepage)
               .appendStr(Myself.Where)
               .appendStr(Myself.Message)
               .appendStr(Myself.ReturnInfo)
            End With

            resp.appendMsg(myData)
            Srv.send(resp)
            Srv.flush()
        End If
    End Sub

    Public Sub HandleResponseMsg(ByVal msg As TipcMsg)
        Dim mver As Short
        Dim m2 As TipcMsg
        Dim resp As TipcMsg
        Dim myData As TipcMsg

        mver = msg.nextInt2
        If mver > messageFormat Then
            frmBoard.DefInstance.wrongMessageFormat("RESPONSE",
mver)
        Else
            m2 = msg.nextMsg
            mver = m2.nextInt2
            frmBoard.DefInstance.newUser(m2)
        End If
    End Sub

    Public Sub HandleUpdateMsg(ByVal msg As TipcMsg)
        Dim mver As Short

        mver = msg.nextInt2
        If mver > messageFormat Then
            frmBoard.DefInstance.wrongMessageFormat("UPDATE",
mver)
        Else
            frmBoard.DefInstance.updateUser(msg)
        End If

    End Sub
```

The TipcErrorEvent event is fired when a SmartSockets error occurs. The delegate that is registered in `initializeClient()` will be called when the event is fired . In this case, the only action is to display relevant error information for the user to acknowledge:

```
Public Sub ErrorHandler(ByVal target As Object,
                        ByVal args As TipcErrorEventArgs)
   If displayErrors Then
        MsgBox("SmartSockets error:  " + args.errNum + ", " +
                    args.errString, MsgBoxStyle.Exclamation +
                    MsgBoxStyle.ApplicationModal +
                    MsgBoxStyle.OKOnly, "SmartSockets Error")
   End If
End Sub
```

The newUser subroutine takes an update message as a parameter, and adds the user information contained within to the Users() array, first removing any old instance of the user. In this example, the number of users is incremented and displayBoard is called to refresh the form display:

```
   Public Sub newUser(ByRef um As TipcMsg)
        Dim j As Object
        Dim thisname As String
        thisname = um.NextStr

        ' if user already on board, remove them
        Dim i As Short
        Dim wasRemoved As Boolean
        i = 0
        While (i < nUsers And Not wasRemoved)
            If Users(i).Who = thisname Then
               For j = i To nUsers - 2
                    Users(j) = Users(j + 1)
               Next j
               nUsers = nUsers - 1
               wasRemoved = True
            End If
            i = i + 1
        End While

        With Users(nUsers)
            .Who = thisname
            .Email = um.NextStr
            .Homepage = um.NextStr
            .Where = um.NextStr
            .Message = um.NextStr
            .ReturnInfo = um.NextStr
        End With

        nUsers = nUsers + 1
        displayBoard()
        showCount()

   End Sub
```

As shown in the next example, the `updateUser` subroutine takes an update message as a parameter, and updates the user information contained within to the `Users()` array. If the user is not currently in the array, they are added. Like the `newUser` subroutine, it calls `displayBoard` to refresh the form display.

```
Public Sub updateUser(ByRef um As TipcMsg)
     Dim i As Object
     Dim thisname As String
     Dim uIndex As Short
     thisname = um.NextStr
     ' user already on board?
     uIndex = -1
     For i = 0 To nUsers - 1
         If Users(i).Who = thisname Then
            uIndex = i
            Exit For
         End If
     Next i

     ' add this user if necessary
     If uIndex = -1 Then
         uIndex = nUsers
         nUsers = nUsers + 1
     End If

     With Users(uIndex)
         .Who = thisname
         .Email = um.NextStr
         .Homepage = um.NextStr
         .Where = um.NextStr
         .Message = um.NextStr
         .ReturnInfo = um.NextStr
     End With

     displayBoard()
     showCount()

End Sub
```

The btnUpdate subroutine, as shown below, is invoked when the Update All button is clicked. It resets the user count and calls doUpdate, re-publishing an announce message. The other users' message board applications will see this and send response messages directly to the running WhoWhere RTclient, populating the Users() array as the message callback events are invoked.

```
Private Sub btnUpdate_Click(ByVal eventSender As System.Object,
                            ByVal eventArgs As
System.EventArgs)
                       Handles btnUpdate.Click
     nUsers = 0
     doUpdate()
     board.Focus()
End Sub
```

Data Flow

The flow of messages used in the WhoWhere program is:

1. Users just joining the message board group (specified by the department configuration field) are announced to the group members by publishing an `mtAnnounce` message containing information specific to the new user.
2. WhoWhere RTclients receive the new user's announcement and reply directly and exclusively to the originating RTclient by publishing an `mtResponse` message containing their information.
3. Upon receiving an `mtResponse` message, a WhoWhere RTclient adds the enclosed user information to their message board.
4. All WhoWhere RTclients receive `mtUpdate` messages, processing and changing the updated information to their message board in real-time.
5. If you click Update All, the board information is discarded and re-built by starting over as if a new client subscribed to the message board.

Notes

There are some points to note when you examine all the WhoWhere source code:

- Although not shown here, the `frmConfiguration` form calls `initializeClient` upon successful completion (that is, when you click OK). This ensures that any changes made to the configuration are accurately reflected by the WhoWhere application's state.
- When WhoWhere is invoked for the first time, the Configuration dialog box appears where you set up your message board environment. Once you enter the information, the Log In window appears.
- There is message format version-control in the WhoWhere program. The first field in every message sent is added with a line similar to this:

```
message.AppendInt2 messageFormat
```

This value is decoded with passages similar to the next example in the message processing events:

```
If mver > messageFormat Then
   frmBoard.DefInstance.wrongMessageFormat("ANNOUNCE", mver)
Else
      …
End If
```

The code extracts the first field from the messages, a two-byte integer, and checks it against the global constant `messageFormat` (see `modGlobals` for the definition of `messageFormat`). This ensures that if newer versions of WhoWhere, with different message grammars are present in the same department, the older RTclients do not corrupt their data with incompatible messages. Your applications may need more sophisticated message version control. This can be implemented with the UserProp property of TipcMsg objects.

- Remember, placing break points in the source code and stepping through the code with Step Into (the F8 key) is the way to see the sequence of events being encountered by the program.
- The global constant Debugging (see modGlobals) can be set to a non-zero value. This enables printing of diagnostic information in the Visual Studio .NET Immediate window, which is useful for grasping the overall operation of the WhoWhere RTclient.
- When you enter your name in the Name field of the Configuration window, enter your first name, followed by your last name (for example, Jane Smith). Names are automatically displayed on the message board in alphabetical order: last name appearing first, followed by a comma and then the first name (for example, Smith, Jane).

Index

N

O

P

R

S

T

U

V

W