

# Programming Windows<sup>®</sup> Services with Microsoft<sup>®</sup> Visual Basic<sup>®</sup> 2008

*Michael Gernaey*

To learn more about this book, visit Microsoft Learning at  
<http://www.microsoft.com/MSPress/books/11309.aspx>

9780735624337

**Microsoft<sup>®</sup>**  
Press

© 2008 Michael Gernaey. All rights reserved.

# Table of Contents

<i>Introduction</i> .....	xvii
<i>Who This Book Is For</i> .....	xvii
<i>How This Book Is Organized</i> .....	xvii
<i>System Requirements</i> .....	xviii
<i>Find Additional Content Online</i> .....	xviii
<i>The Companion Web Site</i> .....	xviii
<i>Support for This Book</i> .....	xix
<i>Questions and Comments</i> .....	xix

## Part I **Defining Windows Services**

<b>1 Writing Your First Service in Visual Basic 2008</b> .....	<b>3</b>
Generating the Project .....	4
Renaming Our Project Files .....	4
Understanding the Wizard Code .....	5
The <OnStart> Method .....	5
The <OnStop> Method .....	5
Other Events .....	6
Writing Our First Code .....	7
Modifying the <OnStart> Method .....	7
Modifying the <OnStop> Method .....	8
Modifying the <OnPause> Method .....	9
Modifying the <OnContinue> Method .....	9
Making the Service Installable .....	10
Setting the Service Properties .....	11
Setting the Startup Options .....	12
Additional Configuration Options .....	12

 **What do you think of this book? We want to hear from you!**

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

[www.microsoft.com/learning/booksurvey/](http://www.microsoft.com/learning/booksurvey/)

Building, Installing, and Deploying . . . . .	12
Creating Your Service Storage Location . . . . .	13
Verifying That You Have .NET 2.0 Installed . . . . .	13
Verifying That Your Service Is Installed . . . . .	14
<OnStart> Message . . . . .	14
<OnPause> Message . . . . .	14
<OnContinue> Message . . . . .	14
<OnStop> Message . . . . .	15
Summary . . . . .	15
<b>2 Expanding Your Service with Threads . . . . .</b>	<b>17</b>
Cleaning Up the Service from the Previous Chapter . . . . .	17
Understanding Threads . . . . .	18
Determining How Many Threads to Create . . . . .	18
Thread Synchronization . . . . .	18
Creating Threads . . . . .	19
Thread Methods . . . . .	19
The New Code. . . . .	20
Thread Funtion Code . . . . .	20
Event-Logging Code. . . . .	21
Updating the <OnStart> Method . . . . .	22
Updating the <OnStop> Method . . . . .	22
Updating the <OnPause> Method . . . . .	22
Updating the <OnContinue> Method . . . . .	23
Updating the Thread Method. . . . .	23
Executing the Thread . . . . .	23
Updating <OnStart> . . . . .	24
Install and Test Your Service . . . . .	24
What Is Thread Cleanup? . . . . .	25
Thread Cleanup Availability. . . . .	25
Threads and Accessibility. . . . .	25
A Problem with the Current <OnStart> Thread . . . . .	25
Thread Cleanup. . . . .	26
Making Thread Cleanup Useful . . . . .	28
Keeping the Thread Alive . . . . .	29
Install and Verify . . . . .	30

Extending <OnPause> and <OnContinue> .....	30
Ways to Control Thread Processing .....	30
Updating <OnPause> .....	30
Updating <OnContinue> .....	31
Using Thread State Control .....	32
Updating <OnPause> .....	33
Updating <OnContinue> .....	34
Updating <OnStop> .....	34
Updating <ThreadFunc> .....	35
Importance of the THREAD_WAIT Value .....	36
Summary .....	37
<b>3 Services and Security .....</b>	<b>39</b>
Security Privileges and Services .....	39
Service Account Security .....	40
Local System .....	41
User Accounts .....	41
Securing the Service .....	43
Protecting Data .....	43
Summary .....	46
<b>Part II Creating Interactive Windows Services</b>	
<b>4 Services and Polling .....</b>	<b>49</b>
Polling the File System .....	49
Adding a Module File .....	50
Adding Event Log InstanceIds .....	50
Adding New Polling Code .....	50
Introduction to Instrumenting a Resource File .....	51
Updating the Service Events .....	52
Modifying Our <OnStart> .....	52
Modifying <OnStop> .....	53
Modifying <OnPause> .....	53
Modifying <OnContinue> .....	54
Writing a New Thread Method .....	54
Install and Verify .....	55

Monitoring with Multiple Threads . . . . .	56
Expanding Processing . . . . .	56
Creating the Code. . . . .	56
Installation and Verification. . . . .	58
Extending the Threading Model . . . . .	58
Monitoring More Than One Folder . . . . .	58
Monitoring More Than One File Type Per Folder . . . . .	59
Outputting to More Than One Folder. . . . .	59
Processing More Than One File Type Per Folder . . . . .	59
When Complexity Steps In . . . . .	60
Adding a FileWorker Class. . . . .	60
Designing a New Class File . . . . .	60
Implementing the Worker Class. . . . .	62
Adding a FileWorker Collection . . . . .	62
Adding New File Type and Input Locations . . . . .	62
Creating the <i>FileWorkerOptions</i> Class . . . . .	63
Updating the FileWorker Constructor . . . . .	64
Updating Our <code>&lt;FileWorker.ThreadFunc&gt;</code> . . . . .	64
Updating <code>&lt;Tutorials.ThreadFunc&gt;</code> . . . . .	65
Updating the <code>&lt;OnStop&gt;</code> Method . . . . .	66
Installation and Verification. . . . .	67
Using Configuration Files . . . . .	68
Application-Specific Configuration File. . . . .	68
The Application-Processing Configuration File . . . . .	69
Updating <code>&lt;Tutorials.ThreadFunc&gt;</code> . . . . .	70
Installation and Verification. . . . .	72
Summary . . . . .	72
<b>5 Processing and Notification . . . . .</b>	<b>73</b>
SMTP Notifications. . . . .	73
File Processing. . . . .	76
Configuring Our New SMTP Class . . . . .	76
Updating the <i>FileWorkerOptions</i> Class . . . . .	77
Updating the <i>FileWorker</i> Class . . . . .	77
Updating the <i>Tutorials</i> Class . . . . .	80
Installation and Verification. . . . .	82

Advanced Processing . . . . .	82
Exploring Processing Options . . . . .	83
Optimizing Processing . . . . .	83
Implementing a Solution . . . . .	85
Creating a New <i>&lt;FileWorker.ProcessingIncoming&gt;</i> Method . . . . .	85
Updating the <i>&lt;Tutorials.OnStop&gt;</i> Method . . . . .	89
Queueing E-mail Notifications . . . . .	89
Decoupling Notifications Implementation Questions . . . . .	90
Decoupling: An Example . . . . .	90
SMTP Queueing Solution . . . . .	91
Updating the <i>SMTP</i> Class . . . . .	94
Updating the <i>FileWorkerOptions</i> Class . . . . .	97
Updating the <i>&lt;Tutorials.ThreadFunc&gt;</i> Method . . . . .	98
Updating the <i>&lt;FileWorker&gt;</i> Constructor . . . . .	99
Installation and Verification . . . . .	100
Summary . . . . .	101
<b>6 User Input, Desktop Interaction, and Feedback . . . . .</b>	<b>103</b>
Understanding Service Feedback . . . . .	103
Configuring a Service to Interact with the Desktop . . . . .	104
Getting Started with Creating the Interactive Service . . . . .	104
Creating a Feedback Form . . . . .	104
Making the Form Visible to the Service . . . . .	106
Updating the <i>FileWorker</i> Class . . . . .	106
Install, Configure, and Verify . . . . .	110
Summary . . . . .	111
<b>7 Data Logging: Processing and Storing Data in SQL Server 2005 . . . . .</b>	<b>113</b>
Configuring Microsoft SQL Server . . . . .	114
Creating a Tutorials Database . . . . .	114
Creating a Users Table . . . . .	114
Creating a User Stored Procedure . . . . .	114
Understanding a LINQSQL Class . . . . .	115
Using LINQ To SQL . . . . .	115
Creating a SQL Class . . . . .	118
Updating the <i>FileWorker</i> Class . . . . .	120
Updating <i>&lt;FileWorker.ProcessFiles&gt;</i> . . . . .	120
Install and Verify . . . . .	122

Data Tracking Validation . . . . .	123
Creating Process Error Folder . . . . .	123
Error Processing Solution. . . . .	123
Updating the <i>FileWorkerOptions</i> Class . . . . .	123
Updating Our Configuration.xml File . . . . .	124
Updating the <i>FileWorker</i> Class . . . . .	124
Updating the <code>&lt;Tutorials.ThreadFunc&gt;</code> Method. . . . .	125
Implementing the Record Failure Code . . . . .	126
Adding a Process <i>Failure</i> Method . . . . .	126
The Worker Thread. . . . .	128
Install and Verify . . . . .	130
Data Migration from One Data Store to Another Data Store . . . . .	130
Creating the Back-End Data Store . . . . .	131
Creating a New Connection String . . . . .	131
Creating a New <code>&lt;ProcessRecords&gt;</code> Method. . . . .	136
Install and Verify . . . . .	139
Reporting Processing Failures. . . . .	139
Optimizing the <i>LINQSQL</i> Class. . . . .	140
Install and Verify . . . . .	141
Summary . . . . .	141

## Part III **Services That Support IT and the Business**

<b>8 Monitoring and Reporting with WMI . . . . .</b>	<b>145</b>
Using WMI with Services. . . . .	145
WMI Architecture . . . . .	146
Creating the Generic WMI Class. . . . .	147
Understanding WMI Classes and Their Uses . . . . .	149
Specific WMI and Custom Classes . . . . .	149
Using the WMI Class . . . . .	151
Adding New EventLog Constants. . . . .	151
Updating <code>&lt;Tutorials.ThreadFunc&gt;</code> . . . . .	151
Adding the WMI Property Reader Method . . . . .	153
Extending the WMI Implementation. . . . .	154
Extending the WMI Class. . . . .	154
Creating the <i>WMIWorkerOptions</i> Class. . . . .	159
Creating the Configuration File . . . . .	160
WMI Service Account. . . . .	160

WMI System Monitoring .....	161
Updating the Configuration File .....	161
WMI <i>Win32_Process</i> Usage .....	161
Updating the <i>WMIWorkerOptions</i> Class .....	163
Updating the <i>&lt;Tutorials.ThreadFunc&gt;</i> Method .....	163
Updating the <i>&lt;Query&gt;</i> Configuration Value .....	164
Updating the <i>&lt;WMI.ProcessWMIRequest&gt;</i> Method .....	164
Service Function Validation .....	165
Service Notification.....	165
Updating the Configuration.xml File .....	166
Updating the WMI Class.....	166
Updating the <i>WMIWorkerOptions</i> Class .....	166
Updating the <i>&lt;Tutorials.ThreadFunc&gt;</i> Method .....	167
Updating the <i>&lt;WMI.ProcessWMIRequest&gt;</i> Method .....	168
Service Validation .....	169
Summary.....	169
<b>9 Talking to the Internet .....</b>	<b>171</b>
Reading and Parsing ASP Pages .....	171
Creating the ASP Master Page.....	172
Calling the ASP Master Page.....	172
Application Log InstanceIds.....	177
Storing ASP Page URL Monitoring Status .....	177
Creating LINQ To SQL Dependencies.....	179
Updating the Configuration File.....	180
Updating the <i>&lt;Tutorials.TThreadFunc&gt;</i> Method .....	181
Updating the Tutorials <i>&lt;OnStop&gt;</i> Method.....	182
Service Validation .....	183
Adding a Dynamic Status ASPX Page.....	183
Creating a New ASP .NET Web Application .....	183
Validating the Web Site .....	186
FTP and Your Service .....	186
Using FTP in the Service.....	187
Creating FTP Directories.....	187
Adding an FTP Class .....	187
Modifying the Configuration File .....	196
Modifying the <i>&lt;Tutorials.ThreadFunc&gt;</i> Method .....	196

	Updating the Tutorials <OnStop> Method .....	197
	Service Validation .....	198
	Uploading Data Using FTP .....	199
	Updating the FTP Class .....	199
	Updating the Configuration File.....	201
	Updating the <Tutorials.ThreadFunc> Method.....	201
	Updating the FTP Class .....	201
	Updating the <Start> Method.....	202
	Service Validation .....	202
	Summary .....	203
<b>10</b>	<b>Services That Listen .....</b>	<b>205</b>
	Listening with TCP/IP .....	205
	Design Points for Service Listeners.....	206
	Creating the First Listener Service .....	206
	Coding the Service Listener.....	206
	Creating a Listener Class .....	207
	Listener Processing Methods .....	215
	Updating the <Tutorials.ThreadFunc> Method.....	216
	Updating the Tutorials <OnStop> Method .....	217
	Service Validation.....	218
	The Test Client.....	218
	Allowing Multiple Connections .....	221
	Extending the <i>Listener</i> Class .....	221
	Updating the <StartListener> Method .....	223
	Service Validation .....	224
	Summary .....	225
<b>11</b>	<b>Advanced Security Considerations and Communications .....</b>	<b>227</b>
	What Does Securing the Service Mean?.....	227
	Service Logon Privileges .....	228
	Securing Your Service's Configuration.....	228
	Options for Securing Configuration Data.....	229
	A Closer Look at Security Options .....	230
	Services as Clients.....	235
	Securing the HTTP Client Service.....	235
	Securing the FTP Service .....	241

Securing the SMTP Client Class .....	247
Writing Secure Code .....	251
Securing In-Memory and On-Disk Data .....	253
Using SSL with Server Services .....	254
Updating the Test Client to Use SSL .....	261
Summary .....	263

## Part IV **Advanced Windows Services Topics**

<b>12 Scheduling, Configuring, Administering, and Setting Up Windows Services .....</b>	<b>267</b>
What Does Scheduling Mean? .....	267
Scheduling Options .....	268
Permission Requirements .....	268
Determining the Type of Scheduling to Use .....	268
Administration of Services .....	269
Types of Configuration Data .....	270
Advanced Service Administration .....	274
Installing Services .....	277
Adding the Setup Project .....	277
Summary .....	279
<b>13 Debugging and Troubleshooting Windows Services .....</b>	<b>281</b>
Debugging Services .....	281
Using the Visual Studio IDE .....	281
Writing Your Service as a Console Application .....	282
Troubleshooting and Monitoring Services .....	282
Task Manager .....	282
Performance Monitor .....	285
Performance Counter Consumers .....	285
Using Perfmon as a Performance Counter Consumer .....	285
Standard System Exposed Performance Counters .....	286
Viewing Perfmon .....	286
Examples of Debugging and Monitoring Your Service .....	287
High CPU .....	289
Summary .....	292

<b>14</b>	<b>Adding Performance Counters</b> . . . . .	<b>293</b>
	Types of Performance Counters . . . . .	293
	Operating System-Exposed Counters . . . . .	293
	Application-Specific Counters . . . . .	294
	Adding Counters to your Service . . . . .	295
	Creating Your Counters . . . . .	295
	Implementing Our Counters in Code . . . . .	298
	Creating Instances of Counters . . . . .	298
	Updating Counter Values . . . . .	299
	Sample Service with Performance Counters . . . . .	300
	Service Validation . . . . .	302
	Summary . . . . .	304
 <b>Part V Appendices</b>		
<b>A</b>	<b>Microsoft Internet Information Server (IIS)</b> . . . . .	<b>307</b>
	Installing Microsoft IIS . . . . .	307
	Installing IIS on Microsoft Vista Ultimate . . . . .	308
	Installing IIS on Windows XP . . . . .	308
<b>B</b>	<b>Microsoft File Transfer Protocol Service</b> . . . . .	<b>311</b>
	Installing Microsoft FTP Service . . . . .	311
	Installing the FTP Service on Windows Server 2003 . . . . .	311
	Installing the FTP Service on Windows Vista Ultimate . . . . .	311
	Installing FTP Services on Windows XP . . . . .	312
<b>C</b>	<b>Microsoft SMTP Service</b> . . . . .	<b>313</b>
	Installing SMTP Services on Windows XP . . . . .	313
	Index . . . . .	315

 **What do you think of this book? We want to hear from you!**

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

[www.microsoft.com/learning/booksurvey/](http://www.microsoft.com/learning/booksurvey/)

# Services That Listen

<b>In this chapter:</b>	
<b>Listening with TCP/IP</b> .....	<b>205</b>
<b>Service Validation</b> .....	<b>218</b>
<b>Allowing Multiple Connections</b> .....	<b>221</b>
<b>Summary</b> .....	<b>225</b>

Sometimes you need your service to be triggered by more than the generation of external data. Services also have the ability to wait for direct input, whether by listening on a Transmission Control Protocol (TCP), User Datagram Protocol (UDP), Simple Network Management Protocols (SNMP), Named Pipes, MailSlots, or any other mechanism that allows for direct one-to-one or one-to-many communications.

You can design your service for more than just processing data. In some cases an end user or even another process needs to request data in real time. To do this, we have to design how the service will process incoming requests or monitor for incoming requests. Depending on how the request or connection is made, we need to set up our service listening protocols appropriate to that communication method. You'll need to decide whether to use static connections—such as a TCP connection—or a call/caller/callback method. For example, when a UDP request comes in, an acknowledgement might be sent, the data gathered, and then a call made from the service to the requester with either a static connection request or with a request for the original caller. Sometimes a request can come in that defines the information being requested and where and how it should be returned to the requester. Your service might be designed to accept a set of parameters from the caller that will determine the information requested, as well as the directory to store the data when the processing is completed. You can also provide security credentials, logon information, IP addresses, and any other information required to process the caller's request.

## Listening with TCP/IP

When using TCP/IP to accept incoming connections and requests, you must account for several things before designing your service. These depend on the solution you are trying to build or the problem you are trying to resolve.

## Design Points for Service Listeners

Many different types of service listeners are available. Whether the protocol is TCP, HTTP, UDP, FTP, or some custom variation, you need to consider some important questions before you start coding your service, including the following:

- Which server port (or list of ports) will requests come in on?
- How many active connections will you allow at one time?
- What type of security will you implement? Will you use network authentication, implied security, or clear-text authentication with user names and passwords stored in a secured place such as Microsoft SQL Server?
- If your service must perform actions on behalf of that caller, will it do so in the context of the caller, or in its own security context? Will your service have more or less security authorization than the caller?
- Will the connections be synchronous or asynchronous?
- Will connections have a time limit?
- Will each request from a caller require a new connection or can connections be static after a user is connected successfully?
- How will the service handle invalid connection attempts?
- In what format does the service expect the requests?
- What format will you use for communications between the service and the client?

## Creating the First Listener Service

The preceding list shows some important characteristics of a service that you must carefully consider before design. In this section, we'll use a single server port to listen for connections. When connections arrive, they will be authenticated using a very simple, basic text authentication scheme. The user name and password will be hard-coded for now. When the connection is authenticated and a secondary socket has been created to service clients' requests, we'll wait for requests to come in from the client. All of these requests will be standard text-based requests with a standard delimiter. When the request comes in, the service will process the request and then return the information to the caller. After the caller acknowledges receipt of the information or the request times out, the connection will be dropped and the resources for that connection and any work it did will be cleaned up.

## Coding the Service Listener

We'll continue with the code in Chapter 9, "Talking to the Internet." The code base gives us our standard service framework with the ability to start, stop, pause, continue, and shut down the service.

## Adding a Configuration File

We need to add a configuration file that our application can use to create single or multiple listeners. In the first example, shown in Listing 10-1, we'll only use a single listener.

### Listing 10-1 Configuration file schema.

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
  <Listeners>
    <Listener>
      <Port>15000</Port>
      <MaxConnections>1</MaxConnections>
    </Listener>
  </Listeners>
</Configuration>
```

The configuration file will allow us to have as many listeners as we want. The listeners themselves consist of the following two properties:

- *Port* represents the server side port to listen on. For most systems this is a number between 1 and 65,000. Ensure that the port you want to listen on is not already in use.
- *MaxConnections* represents how many client connections you can have at one time. Remember that connections do not stay connected on the server port that the client initially connected to. You have to create a server-side socket to hold the client's connection.

## Creating a Listener Class

We need to create a class that can support multiple listeners. Although we won't be creating multiple services, we will be creating the ability for multiple entry points into this service. We could also extend our configuration file to include information that would tell the *Listener* class instance to do a specific task. If a single service could have multiple actions, you would want to have separate server ports, threads, and worker data for each possible action. You could optimize your service even more by separating the workload of each task that your service is capable of. Let's review our *Listener* class, shown in Listing 10-2.

### Listing 10-2 The *Listener* class.

```
Imports System.Threading
Imports System.IO
Imports System.Text
Imports System.Net.Sockets
Imports System.Net
Imports System.ServiceProcess

Public Class Listener
  Public m_Incoming As Thread
  Private m_ThreadAction As ThreadActionState
  Private m_Listener As Socket = Nothing
  Private m_ClientSocket As Socket = Nothing
```

```

Private m_MaxConnections As Integer
Private m_Port As Integer

Public Sub New(ByRef threadaction As ThreadActionState)
    m_ThreadAction = threadaction
End Sub

Public Sub Start()
    m_Incoming = New Thread(AddressOf StartListener)
    m_Incoming.Priority = ThreadPriority.Normal
    m_Incoming.IsBackground = True
    m_Incoming.Start()
End Sub

Private Sub StartListener()
    While Not m_ThreadAction.StopThread
        If Not m_ThreadAction.Pause Then
            Try
                'We need to set up our Port listener and the ability
                'to accept an incoming call.
                Dim localEndPoint As IPEndPoint = Nothing

                m_Listener = New Socket(AddressFamily.InterNetwork, _
                    SocketType.Stream, ProtocolType.Tcp)

                Dim ipHostInfo As IPHostEntry = Dns.GetHostEntry(Dns.GetHostName())
                Dim ipAddress As IPAddress = ipHostInfo.AddressList(0)

                localEndPoint = New IPEndPoint(ipAddress.Any, Me.Port)

                m_Listener.Bind(localEndPoint)
                m_Listener.Listen(Me.MaxConnections)

                Dim bytes() As Byte = New [Byte](1024) {}

                While True
                    ' Program is suspended while waiting for an incoming connection.
                    m_ClientSocket = m_Listener.Accept

                    Dim data As String = Nothing
                    Dim bError As Boolean = False
                    ' An incoming connection needs to be processed.
                    While True
                        Dim iStart As Long = Now.Ticks
                        'Create a Byte Buffer to receive data on.
                        bytes = New Byte(1024) {}

                        Dim bytesRec As Integer = _
                            m_ClientSocket.Receive(bytes)
                        data += Encoding.ASCII.GetString(bytes, 0, _
                            bytesRec)

                        If ((Now.Ticks - iStart) / 10000000) > 30 Then
                            'We have timed out based on a 30 second timeout
                            Try

```

```

        m_ClientSocket.Shutdown(SocketShutdown.Both)
        Catch ex As Exception
            'do nothing
        End Try

        Try
            m_ClientSocket.Close()
        Catch ex As Exception
            'do nothing
        End Try

        Exit While
    End If

    'if we have not timed out yet then let us see if a command
    'has come in and process it
    If Data.IndexOf("<EOF>") > -1 Then
        'If we have found an EOF then we need to process that information
        'we could reset our timeout variable also if we have a command so it
        'does not time out falsely
        'Process the Command
        Dim pszOut As String = Nothing

        Try
            WriteLogEvent(Data, 15, _
                EventLogEntryType.Information, My.Resources.Source)
            Call ProcessCommand(Data, pszOut)
            m_ClientSocket.Send(Encoding.ASCII.GetBytes(pszOut), _
                Encoding.ASCII.GetBytes(pszOut).Length, SocketFlags.None)
        Catch ex As Exception
            Exit While
        End Try
        'clean up
        Try
            m_ClientSocket.Shutdown(SocketShutdown.Both)
        Catch ex As Exception
            End Try

        Try
            m_ClientSocket.Close()
        Catch ex As Exception
            End Try
        End If

        Exit While
    End While

    End While

    Catch nex As SocketException
        WriteLogEvent(My.Resources.ThreadErrorMessage + "_" + _
            nex.ToString + "_" + Now.ToString, THREAD_ERROR, _
                EventLogEntryType.Error, My.Resources.Source)
    Catch tab As ThreadAbortException
        WriteLogEvent(My.Resources.ThreadAbortMessage + "_" + _
            tab.ToString + "_" + Now.ToString, THREAD_ABORT_ERROR, _
                EventLogEntryType.Error, My.Resources.Source)

```

```

        Catch ex As Exception
        WriteLogEvent(My.Resources.ThreadErrorMessage + "_" + _
            ex.ToString + "_" + Now.ToString, THREAD_ERROR, _
            EventLogEntryType.Error, My.Resources.Source)
        End Try
    End If

    If Not m_ThreadAction.StopThread Then
        Thread.Sleep(THREAD_WAIT)
    End If
End While

End Sub

Private Shared Sub WriteLogEvent(ByVal pszMessage As String, _
    ByVal dwID As Long, ByVal iType As EventLogEntryType, _
    ByVal pszSource As String)
    Try
        Dim eLog As EventLog = New EventLog("Application")
        eLog.Source = pszSource

Dim eInstance As EventInstance = New EventInstance(dwID, 0, iType)
        Dim strArray() As String

        ReDim strArray(1)
        strArray(0) = pszMessage
        eLog.WriteEvent(eInstance, strArray)

        eLog.Dispose()
    Catch ex As Exception
        'Do not Catch here as it doesn't do any good for now
    End Try
End Sub

Private Function ProcessCommand(ByVal pszCommand As String, _
    ByRef pszOut As String) As Boolean
    Try
        If pszCommand Is Nothing Then
            Return False
        End If

        'Get the Data and clear out our ending delimiter
        Try
            pszCommand = pszCommand.Remove(pszCommand.Length - 5,
                "<EOF>".Length)
        Catch ex As Exception
            Return False
        End Try

        'Split the Command and find out which one we are doing
        Dim pszArray() As String = Split(pszCommand, "##")

        Select Case UCase(pszArray(0))
            Case "GETDATETIME"
                pszOut = GetDateTime()
        End Select
    End Try
End Function

```

```
Case "GETSERVICESTATUS"  
    pszOut = GetServiceStatus(pszArray(1))  
Case "GETPROCESSLIST"  
    pszOut = GetProcessList()  
Case Else  
    pszOut = Nothing  
    Return False  
End Select  
  
Return True  
Catch ex As Exception  
    Return False  
End Try  
End Function  
  
Private Function GetDateTime() As String  
    Try  
        Return Now.ToString  
    Catch ex As Exception  
        Return Nothing  
    End Try  
End Function  
  
Private Function GetServiceStatus(ByVal pszService As String) As String  
    Try  
        Dim tmpService As New ServiceController(pszService)  
        Dim pszOut As String = Nothing  
  
        Select Case tmpService.Status  
            Case ServiceControllerStatus.ContinuePending  
                pszOut = "ContinuePending"  
            Case ServiceControllerStatus.Paused  
                pszOut = "Paused"  
            Case ServiceControllerStatus.PausePending  
                pszOut = "PausePending"  
            Case ServiceControllerStatus.Running  
                pszOut = "Running"  
            Case ServiceControllerStatus.StartPending  
                pszOut = "StartPending"  
            Case ServiceControllerStatus.Stopped  
                pszOut = "Stopped"  
            Case ServiceControllerStatus.StopPending  
                pszOut = "StopPending"  
            Case Else  
                pszOut = "Unknown"  
        End Select  
  
        Try  
            tmpService.Close()  
            tmpService.Dispose()  
            tmpService = Nothing  
        Catch ex As Exception  
            'Do nothing  
        End Try  
    End Try  
End Function
```

```

        Return pszOut
    Catch ex As Exception
        Return "Unknown"
    End Try
End Function

Private Function GetProcessList() As String
    Try
        Dim pszOut As String = Nothing
        Dim tmpProcesses() As Process = Process.GetProcesses
        Dim objProcess As Process
        For Each objProcess In tmpProcesses
            If pszOut Is Nothing Then
                pszOut = objProcess.ProcessName
            Else
                pszOut += "##" + objProcess.ProcessName
            End If
        Next

        objProcess = Nothing
        tmpProcesses = Nothing

        Return pszOut
    Catch ex As Exception
        Return Nothing
    End Try
End Function

Public Property Port() As Integer
    Get
        Return m_Port
    End Get
    Set(ByVal value As Integer)
        m_Port = value
    End Set
End Property

Public Property MaxConnections() As Integer
    Get
        Return m_MaxConnections
    End Get
    Set(ByVal value As Integer)
        m_MaxConnections = value
    End Set
End Property

Public ReadOnly Property Incoming() As Thread
    Get
        Return m_Incoming
    End Get
End Property
End Class

```

The following sections review this code.

## Listener Class Properties

*Listener* has two properties that we read from our configuration file. First is the *Port* property, which tells us which server port to use for this instance. Second is the *MaxConnections* property, which tells us how many clients can connect to this instance at one time. Each property must be set before the `<Start>` method of the class instance is called.

## The `<StartListener>` Method

If you've never worked with sockets and TCP/IP before, it's especially important that you review this code.

The first thing I do is create an endpoint, shown in Listing 10-3. An endpoint defines the binding information used by the socket to bind to the local server and socket instance based on the port and server IP address.

**Listing 10-3** Define local endpoint used by listener socket.

```
Dim localEndPoint As IPEndPoint = Nothing
```

Next I create a listener socket, shown in Listing 10-4. The listener socket waits on the endpoint for incoming requests.

**Listing 10-4** Define listener socket used by service.

```
m_Listener = New Socket(AddressFamily.InterNetwork, _  
                        SocketType.Stream, ProtocolType.Tcp)
```

I am using the standard TCP protocol. This indicates that I want to create a connection-based socket.

Next, as shown in Listing 10-5, I create the required *IPHostEntry* and *IPAddress* instances that are used by *IPEndPoint* to create the binding information for the listener socket. I use the *Dns.GetHostName* method to get the list of IP addresses of the local computer. I actually get back a list of IP addresses, but I only care about the first one. I could, of course, iterate through the list if I had multiple adapters and wanted to bind to a specific adapter.

**Listing 10-5** Define listener socket attributes used to bind to server local port.

```
Dim ipHostInfo As IPHostEntry = Dns.GetHostEntry(Dns.GetHostName())  
Dim ipAddress As IPAddress = ipHostInfo.AddressList(0)  
localEndPoint = New IPEndPoint(ipAddress, Me.Port)  
m_Listener.Bind(localEndPoint)  
m_Listener.Listen(Me.MaxConnections)
```

Last, you will see that I use *IPEndPoint* to bind the listener socket, and then I use the *Socket.Listen* method to start listening for incoming connections. You should also notice that I am using the *MaxConnections* property from the configuration file to tell the listener how many client sockets can be connected at any time before it will return an unavailable connection error to the clients.

## The <StartListener> Processing Loop

Once we have the service listener socket running, we are waiting for a client connection request. When a request comes in, we want to process that request. Let's review the code that does this, shown in Listing 10-6.

**Listing 10-6** The <StartListener> processing loop.

```
Dim bytes() As Byte = New [Byte](1024) {}
While True
    ' Program is suspended while waiting for an incoming connection.
    m_ClientSocket = m_Listener.Accept

    Dim Data As String = Nothing
    Dim bError As Boolean = False
    ' An incoming connection needs to be processed.
    While True
        Dim iStart As Long = Now.Ticks
        'Create a Byte Buffer to receive data on.
        bytes = New Byte(1024) {}
        Dim bytesRec As Integer = m_ClientSocket.Receive(bytes)
        Data += Encoding.ASCII.GetString(bytes, 0, bytesRec)
        If ((Now.Ticks - iStart) / 10000000) > 30 Then
            'We have timed out based on a 30 second timeout
            Try
                m_ClientSocket.Shutdown(SocketShutdown.Both)
            Catch ex As Exception
            End Try

            Try
                m_ClientSocket.Close()
            Catch ex As Exception
            End Try
            Exit While
        End If
        If Data.IndexOf("<EOF>") > -1 Then
            Dim pszOut As String = Nothing

            Try
                WriteLogEvent(Data, 15, EventLogEntryType.Information,
                    My.Resources.Source)
                Call ProcessCommand(Data, pszOut)
                m_ClientSocket.Send(Encoding.ASCII.GetBytes(pszOut),
                    Encoding.ASCII.GetBytes(pszOut).Length,
                    SocketFlags.None)
            Catch ex As Exception
                Exit While
            End Try

            'clean up
            Try
                m_ClientSocket.Shutdown(SocketShutdown.Both)
            Catch ex As Exception
            End Try

            Try
                m_ClientSocket.Close()
            End Try
        End If
    End While
End While
```

```
        Catch ex As Exception
        End Try
    End If

    Exit While
End While
```

When a client connection request comes in, we use our single client socket and then use the listener socket's *accept* method to assign the client to the client socket. Then we begin an inner loop to receive the request from the client. In this case I am requiring a 30-second window for the client to send its request. If the request doesn't come within the allotted time, I consider a time-out has occurred, and then exit the loop and disconnect the client.

If the client does send its request in the time period allotted, I begin to peek the data and look for the <EOF>, which is required based on the communication specification for this client-server pair.

When I find the <EOF>, the data is read off into the allocated buffer and then sent to the <ProcessCommand> method. (I will go over this method shortly.) If the <ProcessCommand> call has no errors or issues, I use the client socket to send the response to the client, shut down the socket, close the socket, and then go back to listening for another connection request.

## Listener Processing Methods

The *Listener* class has several processing methods that are used to parse, process, and respond to clients' requests. In our service, we support three separate requests. Each request is covered by a separate processing method. Each processing method is wrapped around the <ProcessCommand> function, which takes the request from the client, parses it, calls the processing method, and then returns the data to the <StartListener> thread.

### The <ProcessCommand> Method

<ProcessCommand> is the main method used by our service. The service will use this wrapper method to determine the type of client request and then call the appropriate method to handle the gathering of the client's request. After the request is complete, the <ProcessCommand> method will return the data via a reference pointer to a string passed to it from the <StartListener> thread method. Each processing method will return back the appropriate string to <ProcessCommand>.

### The <GetDateTime> Method

<GetDateTime> will return the current date and time of the local server. Although not an incredibly useful method, <GetDateTime> is good for demonstration purposes.

## The <GetServiceStatus> Method

When <GetServiceStatus> is called, the user will pass in the short name of any service whose status it wants to validate. After this method is called, it will return the state of the requested service. In the case of an error, or if no service is found, <GetServiceStatus> will return an unknown status to the caller.

## The <GetProcessList> Method

<GetProcessList> will return a comma-delimited list to the client of currently running processes on the server. The client can then parse the processes and get an alphabetized list of server processes.

## Updating the <Tutorials.ThreadFunc> Method

We need to update the <ThreadFunc> method (as shown in Listing 10-7) so that we can read in the values from our configuration file.

**Listing 10-7** The <ThreadFunc> method configuration code.

```
Private Sub ThreadFunc()
    Try
        'Load our Configuration File
        Dim Doc As XmlDocument = New XmlDocument()
        Doc.Load(My.Settings.ConfigFile)

        Dim Options As XmlNode
        'Get a pointer to the Outer Node
        Options = Doc.SelectSingleNode("//*[local-name()='Listeners']")

        If (Not Options Is Nothing) Then
            Dim tmpOptions As System.Xml.XPath.XPathNavigator = _
                Options.FirstChild.CreateNavigator()

            If (Not tmpOptions Is Nothing) Then
                Dim children As System.Xml.XPath.XPathNavigator
                Do
                    Try
                        Dim tmpListener As New Listener(m_ThreadAction)

                        children = tmpOptions.SelectSingleNode("MaxConnections")
                        tmpListener.MaxConnections = Int32.Parse(children.Value)

                        children = tmpOptions.SelectSingleNode("Port")
                        tmpListener.Port = Int32.Parse(children.Value)

                        m_WorkerThreads.Add(tmpListener)
                        tmpListener.Start()
                    Catch ex As Exception
                        WriteLogEvent(ex.ToString(), CONFIG_READ_ERROR, _
                            EventLogEntryType.Error, My.Resources.Source)
                    End Try
                Loop While (tmpOptions.MoveNext)
```

```

        End If
    End If

    Catch ex As Exception
        WriteLogEvent(ex.ToString(), ONSTART_ERROR, _
            EventLogEntryType.Error, My.Resources.Source)
        Me.Stop()
    End Try
End Sub

```

As with all our previous services, we need to be able to read in the values from the configuration file and then assign them to the properties of the class instance. In this case, we are assigning both the *MaxConnections* and *Port* properties to the *Listener* class instance. We can have as many instances as we want, and we can listen on practically an unlimited number of ports. I would recommend, however, that you don't use any ports below 1024 because these are usually associated with already existing applications or standards.

## Updating the Tutorials <OnStop> Method

The service <OnStop> method needs to be updated to clean up the service threads properly for each class instance created, which is displayed in Listing 10-8.

**Listing 10-8** The updated <OnStop> service method.

```

Protected Overrides Sub OnStop()
    ' Add code here to perform any tear-down necessary to stop your service.
    Try
        If (Not m_WorkerThread Is Nothing) Then
            Try
                WriteLogEvent(My.Resources.ServiceStopping, ONSTOP_INFO, _
                    EventLogEntryType.Information, My.Resources.Source)

                m_ThreadAction.StopThread = True

                For Each listener As Listener In m_WorkerThreads
                    Me.RequestAdditionalTime(THIRTY_SECONDS)
                    listener.Incoming.Join(TIME_OUT)
                Next
            Catch ex As Exception
                m_WorkerThread = Nothing
            End Try
        End If
    Catch ex As Exception
        'We Catch the Exception
        'to avoid any unhandled errors
        'since we are stopping and
        'logging an event is what failed
        'we will merely write the output
        'to the debug window
        m_WorkerThread = Nothing
        Debug.WriteLine("Error stopping service: " + ex.ToString())
    End Try
End Sub

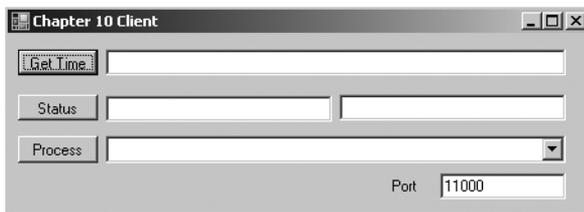
```

## Service Validation

To test the service, we need to connect to the service and call each of the processing methods to see what we get back. For this reason, I have created a client that can use the functionality of the service, whether it is remote or local. The important thing to realize is that we haven't yet implemented any type of authentication in the service, which means that anyone who has access to the server remotely would have access to call the functionality of the service. We will work on this in later chapters.

## The Test Client

I have created a test client that looks like this and can be used to demonstrate the functionality of the service on any given port.



The code, which appears in Listing 10-9, is very simple, but shows how easily you can use your service. The three buttons do three different things: Each creates a Socket, calls `<GetDateTime>`, `<GetProcessList>`, or `<GetServiceStatus>`, and fills in the appropriate box with the response from the server. For the status, the first text box represents the service name, such as `w3svc`, and the second text box represents its status after you click the button and a response returns from the server.

### Listing 10-9 Chapter 10 test client code.

```
Imports System.Text
Imports System.Net
Imports System.Net.Sockets

Public Class frmClient

    Private Sub Button1_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles getTime.Click
        Try
            Dim bytes(1024) As Byte
            Dim ipHostInfo As IPHostEntry = Dns.GetHostEntry(Dns.GetHostName())

            Dim ipAddress As IPAddress

            Dim sendersocket As New Socket(AddressFamily.InterNetwork, _
                SocketType.Stream, ProtocolType.Tcp)
            Dim remoteEP As IPEndPoint

            For Each ipAddress In ipHostInfo.AddressList
                Debug.WriteLine(ipAddress.ToString + "-" +
```

```

IPAddress.AddressFamily.ToString)
    If ipAddress.AddressFamily = AddressFamily.InterNetwork Then
        remoteEP = New IPEndPoint(ipAddress, CInt(txtPort.Text))

        Try
            sendersocket.Connect(remoteEP)
            Exit For
        Catch ex As Exception
            Debug.WriteLine(ex.ToString())
        End Try
    End If
Next

Dim msg As Byte() = _
    Encoding.ASCII.GetBytes("GETDATEIME##<EOF>")
Dim bytesSent As Integer = sendersocket.Send(msg)
Dim bytesRec As Integer = sendersocket.Receive(bytes)
txtTime.Text = Encoding.ASCII.GetString(bytes, 0, bytesRec)

sendersocket.Shutdown(SocketShutdown.Both)
sendersocket.Close()
Catch ex As Exception
End Try
End Sub

Private Sub Button1_Click_1(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Button1.Click
    Try
        cmbProcessList.Items.Clear()
        Dim bytes(1024) As Byte
        Dim ipHostInfo As IPHostEntry = Dns.GetHostEntry(Dns.GetHostName())

        Dim ipAddress As IPAddress = ipHostInfo.AddressList(0)
        Dim remoteEP As New IPEndPoint(ipAddress, CInt(txtPort.Text))
        Dim sendersocket As New Socket(AddressFamily.InterNetwork, _
            SocketType.Stream, ProtocolType.Tcp)

        For Each ipAddress In ipHostInfo.AddressList
            Debug.WriteLine(ipAddress.ToString + "-" +
ipAddress.AddressFamily.ToString)
            If ipAddress.AddressFamily = AddressFamily.InterNetwork Then
                remoteEP = New IPEndPoint(ipAddress, CInt(txtPort.Text))

                Try
                    sendersocket.Connect(remoteEP)
                    Exit For
                Catch ex As Exception
                    Debug.WriteLine(ex.ToString())
                End Try
            End If
        Next

        Dim msg As Byte() = _
            Encoding.ASCII.GetBytes("GETPROCESSLIST##<EOF>")

```

```

        Dim bytesSent As Integer = sendersocket.Send(msg)
        Dim bytesRec As Integer = sendersocket.Receive(bytes)
        Dim tmpArray() As String = Split((Encoding.ASCII.GetString(bytes, 0, bytesRec)),
"##", , CompareMethod.Text)
        Dim iLoop As Integer

        For iLoop = 0 To tmpArray.Length - 1
            cmbProcessList.Items.Add(tmpArray(iLoop))
        Next
        sendersocket.Shutdown(SocketShutdown.Both)
        sendersocket.Close()
    Catch ex As Exception
    End Try
End Sub

Private Sub Button2_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Button2.Click
    Try
        Dim bytes(1024) As Byte
        Dim ipHostInfo As IPHostEntry = Dns.GetHostEntry(Dns.GetHostName())
        Dim ipAddress As IPAddress = ipHostInfo.AddressList(0)
        Dim remoteEP As New IPEndPoint(IPAddress, CInt(txtPort.Text))
        Dim sendersocket As New Socket(AddressFamily.InterNetwork, _
            SocketType.Stream, ProtocolType.Tcp)

        For Each ipAddress In ipHostInfo.AddressList
            Debug.WriteLine(ipAddress.ToString + "-" +
ipAddress.AddressFamily.ToString)
            If ipAddress.AddressFamily = AddressFamily.InterNetwork Then
                remoteEP = New IPEndPoint(ipAddress, CInt(txtPort.Text))

                Try
                    sendersocket.Connect(remoteEP)
                    Exit For
                Catch ex As Exception
                    Debug.WriteLine(ex.ToString())
                End Try
            End If
        Next

        Dim msg As Byte() = _
            Encoding.ASCII.GetBytes("GETSERVICESTATUS##" +
txtService.Text + "<EOF>")
        Dim bytesSent As Integer = sendersocket.Send(msg)
        Dim bytesRec As Integer = sendersocket.Receive(bytes)
        txtServiceStatus.Text =
            Encoding.ASCII.GetString(bytes, 0, bytesRec)

        sendersocket.Shutdown(SocketShutdown.Both)
        sendersocket.Close()
    Catch ex As Exception
    End Try
End Sub
End Class

```

The code in Listing 10-9 is fast and reliable, and you can click the buttons repeatedly. This is neither a complicated process nor a robust one, but you can use this same client to test the code we created previously in this chapter, as we extend the service to allow more than one connection at a time.

If you click one of the buttons, the client utility will connect to the local server and try to run one of the commands. Because it is intended only for demonstration purposes, the error handling is not robust. If you wanted to test it against a remote server, you will need to modify the code to use the name of the server you want to connect to in place of the *Dns.GetHostName()* method used in conjunction with the *Dns.GetHostEntry()* method.

## Allowing Multiple Connections

Allowing only a single connection might work, especially in cases where the requests come from either a single source or at intervals that help ensure that two resources cannot compete for a connection at the same time. However, in some circumstances a single instance of a service could be used to generate requests for multiple resources. A good example of this is when you have a server that is multi-homed for several subnets, and receives requests for data from any or all of these subnets—and possibly from multiple computers or processes on each subnet. In this case, a single point connection would not be beneficial; however, you still want to limit how many connections you allow for performance reasons.

## Extending the *Listener* Class

We are going to modify the code in Listing 10-9 to allow for up to 10 simultaneous connections to our service. This will require us to create up to 10 client sockets to accept the incoming requests, as well as telling our listener socket to accept up to 10 client requests before it returns an error to the client stating that it is unable to connect. Listener sockets only accept one socket at a time. Our code uses synchronous sockets with a backlog of up to 10 socket requests. Remember, though, that our configuration file allows us to specify the maximum number of sockets at any given time. Therefore, it is not a hard-coded value.

We'll use *ThreadPool* to take care of the multiple connections that our class will now be able to accept. Let's review the new modifications.

### The *<SocketThread>* Method

We need to create a secondary thread method that will act as our socket processing thread. Since we will allow more than one connection at a time, having only a single thread won't work. Listing 10-10 shows the newly created thread method.

**Listing 10-10** The *<SocketThread>* thread method.

```
Private Sub SocketThread(ByVal args As Object)
    Dim lSocket As Socket = CType(args, Socket)
```

```

Try
    Dim bytes() As Byte = New [Byte](1024) {}
    Dim Data As String = Nothing
    Dim bError As Boolean = False

    While Not m_ThreadAction.StopThread
        If Not m_ThreadAction.Pause Then
            Dim iStart As Long = Now.Ticks
            bytes = New Byte(1024) {}
            Dim bytesRec As Integer = 1Socket.Receive(bytes)
            Data += Encoding.ASCII.GetString(bytes, 0, bytesRec)

            If ((Now.Ticks - iStart) / 10000000) > 30 Then
                Try
                    1Socket.Shutdown(SocketShutdown.Both)
                Catch ex As Exception
                End Try

                Try
                    1Socket.Close()
                Catch ex As Exception
                End Try

                1Socket = Nothing
                Exit Sub
            End If
            If Data.IndexOf("<EOF>") > -1 Then
                Dim pszOut As String = Nothing
                Try
                    Call ProcessCommand(Data, pszOut)
                    1Socket.Send(Encoding.ASCII.GetBytes(pszOut),
                        Encoding.ASCII.GetBytes(pszOut).Length, SocketFlags.None)
                Catch ex As Exception
                    'clean up
                    1Socket.Shutdown(SocketShutdown.Both)
                    1Socket.Close()
                    1Socket = Nothing
                    Return
                End Try
            End If

            Exit While
        End If
    End While
Catch ex As Exception
    1Socket = Nothing
Finally
    'clean up
    Try
        1Socket.Shutdown(SocketShutdown.Both)
        1Socket.Close()
        1Socket = Nothing
    Catch ex As Exception
        1Socket = Nothing

```

```

        Finally
            !Socket = Nothing
        End Try
    End Try
End Sub

```

The method shown in the preceding listing will be created as a temporary thread by our `<StartListener>` method, which will listen for incoming connections and then create a temporary thread to process the request. The temporary thread accepts a socket as a parameter.

Next, we call the `<ProcessCommand>` method, just as we did earlier, and then send the response to the caller. Last, we close the socket and consider the communication closed. Although this requires us to continually recreate new sockets for the same client calling many different methods, or calling the same method many times, it is for demonstration purposes only. We are not required to drop this socket at all. We could simply make the temporary thread go back into a receive blocking state waiting for a new command from the client.

We could also consider using asynchronous sockets instead of synchronous sockets; however, depending on the workload required, synchronous sockets work for a fast, small service that has limited functional requirements and connections.

## Updating the `<StartListener>` Method

We need to update our primary thread method because we only want it to listen for incoming connections now and not process them. Let's review the new code, shown in Listing 10-11.

**Listing 10-11** The new `<StartListener>` method.

```

Private Sub StartListener()
    While Not m_ThreadAction.StopThread
        If Not m_ThreadAction.Pause Then
            Try
                Dim localEndPoint As IPEndPoint = Nothing

                m_Listener = New Socket(AddressFamily.InterNetwork, _
                    SocketType.Stream, ProtocolType.Tcp)

                Dim ipHostInfo As IPHostEntry =
                    Dns.GetHostEntry(Dns.GetHostName())
                Dim ipAddress As IPAddress = ipHostInfo.AddressList(0)

                localEndPoint = New IPEndPoint(ipAddress.Any, Me.Port)

                m_Listener.Bind(localEndPoint)
                m_Listener.Listen(Me.MaxConnections)

                While Not m_ThreadAction.StopThread
                    Dim tmpSocket As Socket
                    tmpSocket = m_Listener.Accept

                    Dim tmpThread As New Thread(AddressOf SocketThread)

```

```

        tmpThread.IsBackground = True
        tmpThread.Name = "Socket Thread"
        tmpThread.Start(tmpSocket)
    End While
Catch nex As SocketException
    WriteLogEvent(My.Resources.ThreadErrorMessage + "_" +
        nex.ToString + "_" + Now.ToString, THREAD_ERROR,
        EventLogEntryType.Error, My.Resources.Source)
Catch tab As ThreadAbortException
    WriteLogEvent(My.Resources.ThreadAbortMessage + "_" +
        tab.ToString + "_" + Now.ToString, THREAD_ABORT_ERROR,
        EventLogEntryType.Error, My.Resources.Source)
Catch ex As Exception
    WriteLogEvent(My.Resources.ThreadErrorMessage + "_" +
        ex.ToString + "_" + Now.ToString, THREAD_ERROR,
        EventLogEntryType.Error, My.Resources.Source)
End Try
End If

If Not m_ThreadAction.StopThread Then
    Thread.Sleep(THREAD_WAIT)
End If
End While
End Sub

```

Now that we have our processing thread, this code just needs to wait for an incoming socket connection and then hand that socket off to our processing thread. It simply creates a local background thread and then passes it the socket that it just accepted from the client. Notice that our error handling is not as exhaustive as it could or should be. You should be very careful when deciding to run any type of service, especially when it comes to error handling and resources.

## Service Validation

Install the new service and make sure that your configuration file has at least two separate entries for ports to listen on, with more than a single connection as its *MaxConnections* property.

Run multiple instances of the test client, assigning different ports to the test client to cover all possible listening server ports you configured. For each client you should receive a response from the server on the specified port. You can even run multiple instances of the client against the same port and each client will still receive its own response. Remember that the *accept* method can only accept one socket connection at a time per server port, so you won't be able to receive a response at the same instant that you do on another—the server has to process the incoming connection and hand it off to the processing thread. Also, remember that your *MaxConnections* property specifies how many connections can wait in the backlog of queued requests to the server. So if you launch, for example, 11 or more clients and expect them all to work at the same time, you'll be disappointed. When more than 10 clients are queued, client connections will start to fail.

## Summary

- Microsoft Visual Basic 2008 has built-in support for sockets that allows for creation of services that can act as a client or a server answering to incoming requests.
- Services written in Visual Basic 2008 can support secured or unsecured socket protocols.
- Microsoft Visual Basic 2008 *System.NET* and *System.NET.Sockets* classes provide a vast amount of support for traditional and nontraditional connection and connectionless protocols.
- To interact properly, client-server applications must be written to a standard understood by both the client and the server.
- The number of sockets within a client or server is limited by several factors, but revolves around system resources such as memory, network hardware, virtual memory, and CPU. You should write applications to scale with worker threads and sockets, not on an unlimited one-to-one basis with simultaneous connection requests.

