Real World .NET Applications

BUDI KURNIAWAN

Apress™

Real World .NET Applications

ISBN (pbk): 1-59059-082-1

Printed and bound in the United States of America 12345678910

Technical Reviewer: Alwi Wijaya

Editorial Directors: Dan Appleman, Gary Cornell, Simon Hayes, Martin Streicher, Karen Watterson, John Zukowski

Assistant Publisher: Grace Wong

Project Managers: Sofia Marchant, Laura Cheu

Copy Editor: Kim Wimpsett

Compositor: Diana Van Winkle, Van Winkle Design Group

Artists: Kurt Krames, Cara Brunk

Indexer: Valerie Perry

Cover Designer: Kurt Krames

Production Manager: Kari Brooks

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.

Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 9th Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 5

Developing an FTP Client Application

FILE TRANSFER IS an important networking task, and File Transfer Protocol (FTP) client applications are still in wide use despite the ever-increasing popularity of the Web. The FTP client application built for this chapter complies with the current standard as specified in RFC959 by the World Wide Web Consortium (`www.w3.org`); you can use it to connect to and engage in file transfer with standard FTP servers. The main purpose of developing this application is to show how to work with sockets in the .NET Framework.

Overview of the Chapter

This chapter starts by presenting a general overview of sockets and continues with FTP and the project itself. In this chapter, you will find the following sections:

- "**Working with Sockets"**: This section introduces sockets and explains how to use the `System.Net.Sockets.Socket` class and other related classes for network programming.

- **"Understanding FTP"**: This section discusses the protocol for file transfer as specified in RFC959.

- **"Creating an FTP Application Step by Step"**: This section covers a simple console application that is similar to the `ftp.exe` program included in the UNIX/Linux or Windows operating systems. This serves as an introduction to the chapter's FTP project.

- **"Implementing the Project"**: This section contains a detailed discussion on the FTP client application with a Graphical User Interface (GUI).

Working with Sockets

A *socket* is an end point of a connection. It is a descriptor that lets an application read from and write to the network. Using sockets, client applications and server applications can communicate by sending and receiving streams of bytes over connections. To send a message to another socket used in a software application, you need to know not only the machine's Internet Protocol (IP) address that hosts the software application but also the software's process identifier in that machine. A unique number, called a *port*, identifies a software process in a machine. Therefore, to send a message from a socket in one application to another socket in another connection, you need to know the machine's IP address and the application's port number.

In the .NET Framework, the `System.Net.Sockets.Socket` class represents a socket. This class is an implementation of the Sockets Application Programming Interface (API), which is also known as the *Berkeley sockets interface*. The Sockets API was developed in the early 80s at the University of California at Berkeley for the 4.1c release of Berkeley Software Distribution (BSD) Unix. This distribution contained an early version of the Internet protocols.

You can use the `System.Net.Sockets.Socket` class as a socket in a server application as well as in a client application. It also allows both synchronous and asynchronous operations. This chapter only covers using the `Socket` class in a client application. For more details on using this class in a server application, you should consult the .NET Framework documentation.

Instantiating a Socket Object

Instantiating a socket object requires you to pass three arguments to its constructor.

```
Public Sub New( _
  ByVal addressFamily As AddressFamily, _
  ByVal socketType As SocketType, _
  ByVal protocolType As ProtocolType _
)
```

`AddressFamily`, `SocketType`, and `ProtocolType` are enumerations that are part of the `System.Net.Sockets` namespace.

An `AddressFamily` member defines the addressing scheme that a `Socket` object uses to resolve an address. For socket applications that will work on the Internet, you use `InterNetwork`.

`SocketType` determines the type of socket. For this FTP client application, you will use the `Stream` type. This type of socket supports two-way, connection-based byte streams.

`ProtocolType` specifies the type of the low-level protocol that the socket uses to communicate. You must use a stream socket with the Transmission Control Protocol (TCP) type and the `InterNetwork` address family.

Therefore, instantiating a `Socket` object for your application requires the following code:

```
Dim mySocket As New Socket(AddressFamily.InterNetwork, _
  SocketType.Stream, ProtocolType.Tcp)
```

The arguments you pass to the constructor are available in the following read-only properties: `AddressFamily`, `SocketType`, and `ProtocolType`.

Connecting to a Remote Server

Once you have a socket instance, you can connect to a remote server using the `Socket` class's `Connect` method. The `Connect` method attempts to connect to a remote server synchronously. It waits until a connection attempt is successful or has failed before releasing control to the next line in the program. Even though this method is easy to use, there is some preliminary work before you can use this method to connect to a remote server. Consider the `Connect` method signature:

```
Public Sub Connect( ByVal remoteEP As EndPoint)
```

It accepts an argument: an instance of `System.Net.EndPoint`. The abstract `EndPoint` class represents a network address and has a subclass: `System.Net.IPEndPoint`. When using the `Connect` method, you typically pass an `IPEndPoint` object containing the IP address and port number of the remote server to which you want to connect. The question will then be, "How do you construct an `IPEndPoint` object for your socket to connect to a remote server?"

Now, look at the `IPEndPoint` class definition. It has two constructors:

```
Public Sub New( ByVal address As Long, ByVal port As Integer)
Public Sub New( ByVal address As IPAddress, ByVal port As Integer)
```

Of these two constructors, the second is usually used because IP addresses are dotted-quad notation such as 129.36.128.44 and, as you soon will see, in the .NET socket programming, it is easier to get an IP address in this notation than a `Long`. However, the two constructors are actually similar. It is just that the remote IP

address in the first constructor is a `Long`, whereas in the second constructor it is a `System.Net.IPAddress` object. Whichever constructor you choose, you need to have an IP address and the port number of the remote server. The port number is usually not a problem because popular services are allocated default port numbers. For instance, HTTP uses port 80, Telnet uses port 25, and FTP uses port 21.

The IP address is not normally directly available because it is easier to remember domain names such as `microsoft.com` or `amazon.com` rather than the IP addresses mapped to them. With this in mind, you need to resolve a domain name to obtain the IP address of the remote server to which you would like to connect. In the event, to obtain an `IPAddress` instance that you can use to connect to a remote server, you need the following two other classes: `System.Net.Dns` and `System.Net.IPHostEntry`.

The `Dns` class is a final class that retrieves information about a specific host from the Internet Domain Name System (DNS)—hence the name `Dns`. It is mainly used for its `Resolve` method to obtain a set of IP addresses mapped to a domain name. The `Resolve` method returns an `IPHostEntry` object that contains an array of IP addresses. To obtain these IP addresses, you use the `IPHostEntry` class's `AddressList` property.

For example, the following code displays all IP addresses mapped to a DNS name:

```
Try
  Dim server As String = "microsoft.com"  'or any other domain name
  Dim hostEntry As IPHostEntry = Dns.Resolve(server)
  Dim ipAddresses As IPAddress() = hostEntry.AddressList

  Console.WriteLine(server & " is mapped to")
  Dim ipAddress As IPAddress
  For each ipAddress In ipAddresses
    Console.WriteLine(ipAddress.ToString())
  Next
Catch e As Exception
  Console.WriteLine(e.ToString())
End Try
```

When run, the code will display all IP addresses mapped to the DNS name `microsoft.com`.

If a DNS name is mapped to more than one IP address, you can use any of those addresses, even though people usually use the first one. The reason for choosing the first one is that a DNS name is often mapped to one IP address only. You can obtain the first IP address mapped to a DNS name using the following code:

```
HostEntry.AddressList(0)
```

What is more important, once you get an `IPAddress` object, is that you can construct an `IPEndPoint` object to connect to a remote server. If the connection is successful, the `Socket` instance will set its `Connected` property to `True`. A programmer often checks the value of this property before performing other operations on the socket instance because a server application can close a connection after a period of time lapses.

To close a connection explicitly when you are done with a socket, you use the `Close` method. Usually, you need to call the `Shutdown` method prior to invoking `Close` to flush all pending data.

Sending and Receiving Streams

After a socket is connected to a remote machine, you can use it to send and receive data. To send data in synchronous mode, you use the `Send` method. You must place the data you send in an array of bytes. There are four overloads of the `Send` method, all of which return an `Integer` indicating the number of bytes sent.

The first overload is the simplest and the easiest to use of the four. It has the following signature:

```
Overloads Public Function Send( ByVal buffer() As Byte ) As Integer
```

where *buffer* is an array of `Byte` containing the data you want to send. Using this overload, all data in the buffer will be sent.

The second overload allows you to send all data in the buffer and specify the bitwise combination of the `System.Net.Sockets.SocketFlags` enumeration members. It has the following signature:

```
Overloads Public Function Send( _
  ByVal buffer() As Byte, _
  ByVal socketFlags As SocketFlags _
 ) As Integer
```

The third overload allows you to send all or part of the data in the buffer and specify the bitwise combination of the `SocketFlags` enumeration:

```
Overloads Public Function Send( _
  ByVal buffer() As Byte, _
  ByVal size As Integer, _
  ByVal socketFlags As SocketFlags _
) As Integer
```

In this overload, *size* is the number of bytes to be sent.

The last overload is similar to the third overload, but it also allows you to specify an offset position in the buffer to begin sending data. Its signature is as follows:

```
Overloads Public Function Send( _
  ByVal buffer() As Byte, _
  ByVal offset As Integer, _
  ByVal size As Integer, _
  ByVal socketFlags As SocketFlags _
) As Integer
```

In this overload, *offset* is the offset position.

To receive data synchronously, you use the `Receive` method. This method also has four overloads that are similar to the `Send` method overloads. The signatures of the overloads are as follows:

```
Overloads Public Function Receive( ByVal buffer() As Byte ) As Integer

Overloads Public Function Receive( _
  ByVal buffer() As Byte, _
  ByVal socketFlags As SocketFlags _
 ) As Integer

Overloads Public Function Receive( _
  ByVal buffer() As Byte, _
  ByVal size As Integer, _
  ByVal socketFlags As SocketFlags _
) As Integer

Overloads Public Function Receive( _
  ByVal buffer() As Byte, _
  ByVal offset As Integer, _
  ByVal size As Integer, _
  ByVal socketFlags As SocketFlags _
) As Integer
```

When using the `Receive` method, you can use the `Socket` class's `Available` property, which specifies the number of bytes of data received and is available to be read.

Understanding FTP

RFC959 specifies the protocol for file transfer and is downloadable from `www.w3.org/Protocols/rfc959/A3_FTP_RFCs.html`. This protocol defines how an FTP client application and an FTP server must communicate.

Just like any client-server application, an FTP server should be available all the time and a server does not know anything about its clients. It is always the client that initiates a connection with the server. Before any file transfer can happen, a client needs to connect to the FTP server and log in with a username and a password. Some FTP servers allow anyone to access some or all of their content by requesting them to log in using an anonymous account—in other words, by using "anonymous" as the username and their email address as the password.

In FTP, two connections need to be established between a client and an FTP server. The first connection, the control connection, remains open during the whole session and acts as the communication channel the client uses to send requests to the server and for the server to send responses to those requests. The second connection is the data connection used to transfer files and other data. This connection opens just before some data need to be transferred. The data connection closes right after the data transfer.

Typically, the "conversation" on the control channel after a connection is established goes like this:

1. **Server**: OK, you are now connected. Tell me what you want.

2. **Client**: I would like to log in. Here is my username: "James."

3. **Server**: Username received. Now, send me your password.

4. **Client**: My password is "s3m1c0nduct0r."

By sending the username and password, the client tries to log in. If the login fails, the server asks the client to send the password again. If it is successful, an FTP session starts and file transfer can begin. The server terminates the session if it does not hear anything from the client within some period of time, usually 900 seconds.

The client can start transferring files between itself and the server. Again, the "conversation" goes like the following:

1. **Client**: Please send me `companySecret.doc`.

2. **Server**: Here you go. I am sending it from port *x*.

The client then uses another socket instance and tries to connect to the IP address and port specified by the server. Once connected at this port, the server starts sending the file. When all the data is sent, the server automatically closes the data connection. Then the server uses the control channel to send the client the following message:

Server: Connection complete.

A data connection also opens when the client needs to transfer a file or when the server needs to send the list of files and subdirectories in the specified directory. What really happens is of course more technical than the previous description. However, you should have the general idea.

Using FTP Commands

In a conversation between an FTP client application and an FTP server, the client sends a series of FTP commands and the server replies to each command sent by the client. The next client operation is determined by the previous server's reply.

Table 5-1 describes the FTP command that a client application can send to the server.

Table 5-1. FTP Commands

COMMAND	TYPE	DESCRIPTION
USER	Access control	Sends the username to the server to log in. This is normally the first command sent by the client after a control connection is established.
PASS	Access control	Sends the user password to the server to log in. This is normally the command that must be sent immediately after the USER command is sent.
ACCT	Access control	Sends the user's account information. Some FTP servers may require the user's account information to log in, and some may not. If required, this command must be sent right after the server sends the response to the client's PASS command. This response also determines whether the ACCT command needs to be sent. If the server sends a 332 cod to the client's PASS command, the ACCT command must be sent.

(Continued)

Table 5-1. FTP Commands (Continued)

COMMAND	TYPE	DESCRIPTION
CWD	Access control	Changes the server's working directory.
CDUP	Access control	Changes to the parent directory. This is a special case for the CWD command.
SMNT	Access control	Mounts a different file system data structure without altering the client's login or accounting information.
REIN	Access control	Terminates the client, flushing all input/output and account information. Upon receipt of this command, the server leaves the control connection open.
QUIT	Access control	Terminates the client and closes the control connection. If file transfer is in progress when this command is received, the connection remains open for the server to send the file transfer completion reply code. Afterward, the connection closes.
PORT	Transfer	Specifies the data port to be used in the data connection. This command is normally not necessary because there are defaults for both the user and server data ports.
PASV	Transfer	Asks the server to become passive—in other words, requests the server to listen on a data port and to wait for a connection rather than initiate one upon receipt of a file transfer command. The server replies by sending the host address and port number this server is listening on for the next file transfer.
TYPE	Transfer	Specifies the representation type—in other words, whether the representation is ASCII, EBCDIC, or Image (binary).
STRU	Transfer	Specifies the file structure.
MODE	Transfer	Specifies the transfer mode (Stream, Block, or Compressed). The default is Stream.
RETR	Service	Requests the transfer of the specified file from the server.
STOR	Service	Asks the server to accept a file from the client. If a file with the same name already exists in the working directory, the file will be overwritten.
STOU	Service	Similar to STOR, but the file is saved in a different name. The new filename is created in the current directory under a name unique to that directory.
APPE	Service	Asks the server to accept a file from the client. If a file with the same name already exists in the working directory, the transferred file will be appended to the existing file.

(Continued)

Table 5-1. FTP Commands (Continued)

COMMAND	TYPE	DESCRIPTION
ALLO	Service	Requests the server to reserve sufficient storage to accommodate the new file to be transferred.
REST	Service	Restarts a new file transfer.
RNFR	Service	Specifies the name of the file to be renamed. This command must be followed immediately by the RNTO command.
RNTO	Service	Specifies the new name for the file to be renamed.
ABOR	Service	Aborts the previous FTP service command.
DELE	Service	Deletes the specified file.
RMD	Service	Removes directory.
MKD	Service	Makes a new directory
PWD	Service	Prints the working directory.
LIST	Service	Requests the list of files/subdirectories in the specified directory and information on each individual file/subdirectory such as file size, modified date, and so on.
NLST	Service	Requests the list of files/subdirectories in the specified directory without information about each individual file/subdirectory.
SITE	Service	Asks the server to provide services specific to its system that are essential to file transfer but not sufficiently universal to be included a command in the protocol.
SYST	Service	Inquires about the server's operating system.
STAT	Service	Inquires about the status of a file transfer.
HELP	Service	Requests some helpful information regarding the server's implementation status over the control connection to the client.
NOOP	Service	No operation.

You terminate each FTP command with a carriage-return line feed. For example, the following code connects to `ftp.microsoft.com` and sends the USER command indicating the username James:

```
Private server As String = "ftp.microsoft.com"
Private port As Integer = 21
Private userName As String = "James"
Private controlSocket As Socket
Try
  controlSocket = New _
    Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp)
  controlSocket.Connect(New IPEndPoint(Dns.Resolve(server).AddressList(0), port))
```

```
  If controlSocket.Connected Then
    Console.WriteLine("Connected. Waiting for reply...")
    Dim bytes(511) As Byte ' an array of 512 bytes
    Dim receivedByteCount As Integer
    ' receive the server's response on successful connect
    receivedByteCount = controlSocket.Receive(bytes)

    ' send the USER command
    Dim command As String
    command = "USER " & userName & ControlChars.CrLf
    controlSocket.Send(Encoding.ASCII.GetBytes(command), command.Length, 0)
  End If
Catch
End Try
```

An FTP connection carries a two-way dialogue between the FTP server and the client application. The FTP server sends a reply to any command sent by the client. To understand how the conversation takes place, it is important to understand the server's replies.

Sending FTP Replies

An FTP server replies to every FTP command from a client. These replies ensure that requests and actions are synchronized during the process of file transfer. They are also useful so that the client always know the state of the server.

An FTP command generates one or more replies. For example, a USER command makes the server send a reply requesting the client to send the PASS command. Some commands generate more replies. An example of multiple replies is when an FTP server is about to send a file to a client. First, the server sends a reply notifying the client that the file transfer process will commence. After the file transfer finishes and the data connection closes, the server sends the second reply telling the client that the file transfer has completed.

An FTP reply consists of a reply code followed by a space and some description. A reply code is always a three-digit number specified in RFC959, but the descriptions can be different from one server implementation to another. An FTP client should only rely on the reply code.

Each of the three digits in a reply code has special meaning. The first digit is the most important and indicates whether the FTP command is successful, has failed, or is incomplete. There are five possible values for the first digit in the reply code: 1, 2, 3, 4, and 5.

Table 5-2 describes the meaning of each possible value of the first digit in a reply code.

Table 5-2. The First Digit in an FTP Reply

VALUE	DESCRIPTION
1	Positive preliminary reply. The requested action is being initiated; expect another reply before proceeding with a new command. This type of reply indicates that the command was accepted and the client may now pay attention to the data connections for implementations where simultaneous monitoring is difficult.
2	Positive completion reply. The requested action has been successfully completed. A new request may be initiated.
3	Positive intermediate reply. The command has been accepted, but the requested action will not be active, pending receipt of further information. The user should send another command specifying this information.
4	Transient negative completion reply. The command was not accepted and the requested action did not take place, but the error condition is temporary and the action may be requested again. The user should return to the beginning of the command sequence, if any. Note that it is difficult to assign a meaning to *transient*, particularly when two distinct sites (server and client) have to agree on the interpretation.
5	Permanent negative completion reply. The command was not accepted and the requested action did not take place. The client is discouraged from repeating the exact request (in the same sequence).

Table 5-3 describes the meaning of each possible value of the second digit in a reply code.

Table 5-3. The Second Digit in an FTP Reply

VALUE	DESCRIPTION
0	These replies refer to syntax errors, syntactically correct commands that do not fit any functional category, and unimplemented or superfluous commands.
1	These are replies to requests for information, such as status or help.
2	These replies refer to the control and data connections.
3	Authentication and accounting. These are replies for the login process and accounting procedures.
4	Unspecified yet.
5	These replies indicate the status of the server file system *vis-à-vis* the requested transfer or other file system action.

The third digit of the reply code gives a finer gradation of the meaning indicated by the second digit. Table 5-4 should make it clearer.

Table 5-4. FTP Reply Codes in Numerical Order

VALUE	DESCRIPTION
110	Restart marker reply.
120	Service ready in *nnn* minutes.
125	Data connection already open; transfer starting.
150	File status OK; about to open data connection.
200	Command OK.
202	Command not implemented, superfluous at this site.
211	System status, or system help reply.
212	Directory status.
213	File status.
214	Help message.
215	NAME system type, where NAME is an official system name from the list in the Assigned Numbers document.
220	Service ready for new user.
221	Service closing control connection.
225	Data connection open; no transfer in progress.
226	Closing data connection. Request file action successful (for example, file transfer or file abort).
227	Entering Passive Mode (h1,h2,h3,h4,p1,p2).
230	User logged in; proceed.
250	Requested file action OK, completed.
257	"PATHNAME" created.
331	Username OK; need password.
332	Need account for login.
350	Requested file action pending further information.
421	Service not available, closing control connection. This may be a reply to any command if the service knows it must shut down.
425	Cannot open data connection.
426	Connection closed; transfer aborted.
450	Requested file action not taken. File unavailable (for example, file busy).
451	Requested action aborted: local error in processing.

(Continued)

Table 5-4. FTP Reply Codes in Numerical Order (Continued)

VALUE	DESCRIPTION
452	Requested action not taken. Insufficient storage space in system.
500	Syntax error, command unrecognized. This may include errors such as command line too long.
501	Syntax error in parameters or arguments.
502	Command not implemented.
503	Bad sequence of commands.
504	Command not implemented for that parameter.
530	Not logged in.
531	Need account for storing files.
550	Requested action not taken. File unavailable (for example, file not found, no access).
551	Requested action aborted: page type unknown.
552	Requested file action aborted. Exceeded storage allocation (for current directory or dataset).
553	Requested action not taken. Filename not allowed.

Usually, the description in a reply code is one line long. There are cases, however, where the text is longer than a single line. In these cases the complete text must be bracketed so the client application knows when it should stop reading the reply. This requires a special format on the first line to indicate that more than one line is coming, and another on the last line to designate it as the last. At least one of these lines must contain the appropriate reply code to indicate the state of the transaction. The RFC959 decides that both the first and last line codes should be the same in the case of multiline reply description.

For example, replying to a PASS command, an FTP server may send the single-line reply:

```
230 User logged in.
```

Another FTP server may send the following multiline reply:

```
230-User logged in. Welcome to the Atlantis Research Center.
Please note that access to this site is restricted to authorized people only.
This site is closely monitored 24 hours a day.
230 Please proceed.
```

Creating an FTP Application Step by Step

To test this application (and the project), you need an FTP server. If you are connected to the Internet, you can use a number of FTP servers that allow anonymous access. These servers normally allow you to download files but not upload files. Alternatively, you can use a local FTP server found in Windows 2000 Server. The following sections start with installing and configuring an FTP server and then continue with a simple FTP client application.

Installing and Configuring an FTP Server in Windows 2000 Server

If you have access to Windows 2000 Server, you are in luck. One of the programs you can install is an FTP server. If it is already installed, you can configure it through the Internet Service Manager whose applet can be found in Administrative Tools in the Control Panel. If it is not yet installed, it is now time to do so.

To install the FTP Server in Windows 2000 Server, follow these steps:

1. Double-click Add/Remove Programs from Control Panel.

2. Click the Add/Remove Windows Components button on the left side of the page. The Windows Components Wizard will display. One of the items displayed in the Components list is Internet Information Services (IIS). Click this item.

3. Click the Details button. The Internet Information Services dialog box shows.

4. Check the File Transfer Protocol (FTP) Server subcomponent.

5. Click the OK button and follow the instructions to install. You will be asked to insert the Windows 2000 Server installation CD.

The main task in the configuration is to map the directory that will become the root of the FTP server. In addition, you can also set the session timeout and the directory list style. Any user who has access to the Windows 2000 server will have the same access to the FTP server.

To configure the FTP server, follow these steps:

1. Double-click Administrative Tools from Control Panel.

2. Double-click the Internet Service Manager icon. You should be able to see the Default FTP Site icon under the machine name.

3. Right-click the Default FTP Site icon and click Properties. The Default FTP Site Properties dialog box will display.

4. Click the Home Directory tab, as shown in Figure 5-1.

5. In the FTP Site Directory section, browse to the directory that you want to be the root of the FTP server.

6. In the FTP Site Directory section, make sure that the Read and Write check boxes are selected.

7. Select UNIX in the Directory Listing Style section.

8. Click the Apply button and then the OK button.

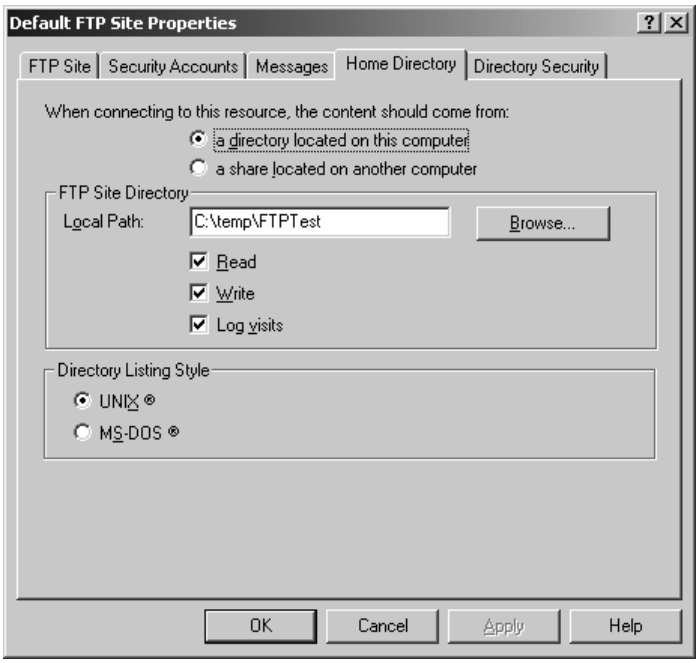

Figure 5-1. The Home Directory tab of the Default FTP Site Properties dialog box

Using the NETFTP Application

The NETFTP console application is a simple FTP client application that is similar to the `ftp.exe` program you can find in Unix/Linux or Windows. It consists of a class, `NETFTP`, that you can find in the `Listings/Ch05/Other/NETFTP.vb` file. The main purpose of this small application is to show how to use the `System.Net.Sockets.Socket` class to connect to an FTP server and do file transfer; it does not worry too much about error handling. After understanding this application, you can understand the project more easily.

To compile this program, type the following command in the command prompt:

```
vbc -r:System.dll FTPNET.vb
```

The result is an executable called `FTPNET.exe`.

The first thing to do to use this program is connect to an FTP server by typing the following:

```
NETFTP <server> <user name> <password>
```

If the connection attempt was successful, a message such as the following will display in the console.

```
Connected. Waiting for reply...

220 bulbul Microsoft FTP Service (Version 5.0).
USER Administrator

331 Password required for Administrator.
PASS
230 User Administrator logged in.

215 Windows_NT version 5.0

200 Type set to A.
```

If the connection failed, an error message will display.

Once connected, you will see the `NETFTP>` prompt. You can type one of the following commands in this prompt: CWD, DELE, LIST, PWD, QUIT, RETR, STOR. Upon execution, the server sends a reply that will display on the console. Other than that, the program outputs the "Command Invalid" message.

Upon successful execution of a command, the program displays the `NETFTP>` prompt again, indicating it is ready to accept a new command.

The valid commands are explained in the following sections.

CWD

This command causes the program to send a CWD command to the FTP server. This command changes the remote working directory. The syntax of this command is as follows:

```
CWD directory
```

where *directory* is the name of the directory to which you want to change. For example, to change the working directory to the `/files/program` directory, type the following:

```
CWD /files/program
```

You can also pass a relative path as the argument. To change to the parent directory of the current directory, type the following:

```
CWD ..
```

If the command successfully executes, the server sends the following message, which displays on the console:

```
250 CWD command successful.
```

If it cannot find the destination directory, the server sends the following message:

```
550 /prog/images: The system cannot find the file specified.
```

DELE

This command causes the application to send a DELE command to the connected FTP server. The DELE command deletes a file in the server. The syntax of this command is as follows:

```
DELE pathToFileToDelete
```

LIST

This command does not take an argument. It causes the application to send a LIST command to the server. The LIST command displays the content of the current directory. For example, the following is the output of the LIST directory:

```
227 Entering Passive Mode (127,0,0,1,6,77).

125 Data connection already open; Transfer starting.
226 Transfer complete.
drwxrwxrwx   1 owner    group               0 Jul 28 21:30 April2001
-rwxrwxrwx   1 owner    group          344064 Jul 29 13:25 Chapter5.doc
-rwxrwxrwx   1 owner    group           12118 Jul 29 13:25 complete.wav
-rwxrwxrwx   1 owner    group           12118 Jul 30 10:05 complete2.wav
-rwxrwxrwx   1 owner    group            1050 Jul 29 13:25 Folder.gif
-rwxrwxrwx   1 owner    group           53248 Jul 28 20:08 FTPClient.exe
drwxrwxrwx   1 owner    group               0 Jul 28 20:21 June 2002
-rwxrwxrwx   1 owner    group            9216 Jul 30 10:02 NETFTP.exe
-rwxrwxrwx   1 owner    group           12077 Jul 30 10:02 NETFTP.vb
drwxrwxrwx   1 owner    group               0 Jul 30 10:05 New Folder
drwxrwxrwx   1 owner    group               0 Jul 26 20:09 New Folder (2)
-rwxrwxrwx   1 owner    group          132717 Jul 29 13:24 rfc959.txt
```

Note that this directory listing is in the Unix style. An FTP server can choose to use another style.

PWD

The PWD command causes the application to send a PWD command to the FTP server. The PWD command displays the current directory. This command does not take an argument. On successful execution, this is an example of what the server may send:

```
257 "/" is current directory.
```

QUIT

The QUIT command causes the application to send a QUIT command to the FTP server. The QUIT command causes the connection to be closed. This command does not take an argument.

RETR

The RETR command causes the application to send a RETR command to the FTP server. The RETR command downloads a file from the server. The syntax of this command is as follows:

```
RETR pathToFileToDownload
```

STOR

The STOR command causes the application to send a STOR command to the FTP server. The STOR command uploads a file in the local directory. The syntax of this command is as follows:

```
STOR pathToFileToUpload
```

How the NETFTP Program Works

The NETFTP program comprises one class: `NETFTP`. The following sections describe the `NETFTP` class.

Declarations

The following is the declarations part of the NETFTP class:

```
Private port As Integer = 21
Private controlSocket As _
  New Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp)
Private dataSocket As Socket
Private serverAddress As String

Public replyMessage As String
Public replyCode As String
```

Note that there are two `Socket` object references used in this class: `controlSocket` and `dataSocket`.

Methods

The following are the methods in the NETFTP class.

Connect

The Connect method connects to an FTP server (see Listing 5-1).

Listing 5-1. The Connect *Method*

```
  Public Sub Connect(ByVal server As String)
    Try
      controlSocket.Connect(New↵
      IPEndPoint(Dns.Resolve(server).AddressList(0), port))
    Catch e As Exception
      Console.WriteLine(e.ToString())
      Return
    End Try
    If controlSocket.Connected Then
      Console.WriteLine("Connected. Waiting for reply...")
      GetResponse()
    Else
      Console.WriteLine("Couldn't connect.")
    End If
  End Sub
```

The Connect method uses the controlSocket's Connect method to connect to an FTP server:

```
Try
  controlSocket.Connect(New IPEndPoint(Dns.Resolve(server).AddressList(0), port))
Catch e As Exception
  Console.WriteLine(e.ToString())
  Return
End Try
```

If the connection attempt is successful, the `controlSocket`'s `Connected` property will be set to `True`. In this case, a message prints on the console and the `GetResponse` method is invoked:

```
If controlSocket.Connected Then
  Console.WriteLine("Connected. Waiting for reply...")
  GetResponse()
```

Otherwise, a "Couldn't connect" message displays on the console:

```
Else
  Console.WriteLine("Couldn't connect.")
End If
```

PassiveData

The `PassiveData` method sends the PASV command to the connected FTP server to make the server become passive (see Listing 5-2).

Listing 5-2. The `PassiveData` *Method*

```
Private Sub PassiveDataConnection()
  SendCommand("PASV" & ControlChars.CrLf)
  GetResponse()
  Dim addr As String = replyMessage

  addr = addr.Substring(addr.IndexOf("("c) + 1, _
    addr.IndexOf(")"c) - addr.IndexOf("("c) - 1)
  Dim address As String() = addr.Split(","c)

  Dim ip As String = address(0) & "." & address(1) & "." & address(2) & "." & _
    address(3)
  Dim port As Integer = Convert.ToInt32(address(4)) * 256 + _
    Convert.ToInt32(address(5))

  dataSocket = New Socket(AddressFamily.InterNetwork, SocketType.Stream,
    ProtocolType.Tcp)
  dataSocket.Connect(New IPEndPoint(IPAddress.Parse(ip), port))
End Sub
```

If the PASV command successfully executes at the server, the server sends the address and port number of the data connection for the client to connect. Therefore, on a successful execution of PASV, the server replies by sending a string of the following format:

```
227 Entering Passive Mode (a1,a2,a3,a4,p1,p2).
```

where `a1`, `a2`, `a3`, and `a4` are parts of an IP address in dotted-quad notation, and `p1` and `p2` signify the port number. Consequently, you can obtain the IP address using the following lines:

```
Dim addr As String = replyMessage
addr = addr.Substring(addr.IndexOf("("c) + 1,↵
addr.IndexOf(")"c) - addr.IndexOf("("c) - 1)
Dim address As String() = addr.Split(","c)
Dim ip As String = address(0) & "." & address(1) & "."↵
& address(2) & "." & address(3)
```

You obtain the port number by multiplying p1 with 256 and adding the result to p2:

```
Dim port As Integer = Convert.ToInt32(address(4)) *↵
256 + Convert.ToInt32(address(5))
```

Then, you instantiate a data socket and invoke its `Connect` method:

```
dataSocket = New Socket(AddressFamily.InterNetwork, SocketType.Stream, _
  ProtocolType.Tcp)
dataSocket.Connect(New IPEndPoint(IPAddress.Parse(ip), port))
```

After the `Connect` method is called, the data socket is readily available to the method that invokes the `PassiveData` method.

GetResponse

The `GetResponse` method receives the byte stream from the connected FTP server. This method is invoked immediately after an FTP command is sent to the server (see Listing 5-3).

Listing 5-3. The `GetResponse` *Method*

```
Private Sub GetResponse()
  ' this method listens for the server response and receives all bytes
  ' sent by the server
  ' A server response can be single line or multiline.
  ' If the fourth byte of the first line is a hyphen, then it is
  ' multiline. If multiline, waits until the line that starts with the
  ' response code (the first three bytes of the first line).

  Dim bytes(511) As Byte ' an array of 512 bytes
  Dim receivedByteCount As Integer
  Dim response As String = ""

  ' get the first line
  receivedByteCount = controlSocket.Receive(bytes)
  response = Encoding.ASCII.GetString(bytes, 0, receivedByteCount)
  Dim multiline As Boolean = (response.Chars(3) = "-"c)

  If multiline Then
    If response.Length > 3 Then
      replyCode = response.Substring(0, 3)
    End If
    Dim line As String = ""
    Dim lastLineReached As Boolean = False
    While Not lastLineReached
      receivedByteCount = controlSocket.Receive(bytes)
      line = Encoding.ASCII.GetString(bytes, 0, receivedByteCount)
      response += line

      If line.IndexOf(ControlChars.CrLf & replyCode & " ") <> -1 Then
        lastLineReached = True
      End If
      If lastLineReached Then
        'just wait until CRLF is reached
        While Not line.EndsWith(ControlChars.CrLf)
          receivedByteCount = controlSocket.Receive(bytes)
          line = Encoding.ASCII.GetString(bytes, 0, receivedByteCount)
          response += line
        End While
      End If
    End While
  Else
```

```
    While receivedByteCount = bytes.Length And _
      Not response.EndsWith(ControlChars.CrLf)
      receivedByteCount = controlSocket.Receive(bytes)
      response += Encoding.ASCII.GetString(bytes, 0, receivedByteCount)
    End While
  End If

  Console.WriteLine()
  Console.Write(response)

  If response.Length > 3 Then
    replyCode = response.Substring(0, 3)
    replyMessage = response.Substring(3, response.Length - 3)
  Else
    replyCode = ""
    replyMessage = "Unexpected Error has occurred."
  End If

End Sub
```

The first thing this method does is to check the fourth byte of the reply to determine whether the server reply is a single-line or a multiline response. The fourth byte in a single-line response is a space, whereas it will be a hyphen in a multiline response:

```
Dim bytes(511) As Byte ' an array of 512 bytes
Dim receivedByteCount As Integer
Dim response As String = ""

' get the first line
receivedByteCount = controlSocket.Receive(bytes)
response = Encoding.ASCII.GetString(bytes, 0, receivedByteCount)
Dim multiline As Boolean = (response.Chars(3) = "-"c)
```

If the response is multilined, the `GetResponse` method reads the reply code (the first three digits of the first line) and keeps reading the response in a `While` loop until the last line is reached. The last line is a line that starts with a reply code:

```
If multiline Then
  If response.Length > 3 Then
    replyCode = response.Substring(0, 3)
  End If
  Dim line As String = ""
```

```
  Dim lastLineReached As Boolean = False
  While Not lastLineReached
    receivedByteCount = controlSocket.Receive(bytes)
    line = Encoding.ASCII.GetString(bytes, 0, receivedByteCount)
    response += line

    If line.IndexOf(ControlChars.CrLf & replyCode & " ") <> -1 Then
      lastLineReached = True
    End If
```

When the last line is reached, it continues reading until the last byte is received:

```
If lastLineReached Then
  'just wait until CRLF is reached
  While Not line.EndsWith(ControlChars.CrLf)
    receivedByteCount = controlSocket.Receive(bytes)
    line = Encoding.ASCII.GetString(bytes, 0, receivedByteCount)
    response += line
  End While
```

If it is a single-line response, it just continues reading until the last byte in the stream is received:

```
Else
  While receivedByteCount = bytes.Length And _
    Not response.EndsWith(ControlChars.CrLf)
    receivedByteCount = controlSocket.Receive(bytes)
    response += Encoding.ASCII.GetString(bytes, 0, receivedByteCount)
  End While
```

The response will then be sent to the console:

```
Console.WriteLine()
Console.Write(response)
```

Next, `replyCode` and `replyMessage` are assigned the reply code and message of the server response, respectively:

```
replyCode = response.Substring(0, 3)
replyMessage = response.Substring(3, response.Length - 3)
```

Login

The `Login` method accepts a username and a password that will be sent as the user's credential to log in to the connected FTP server (see Listing 5-4).

Listing 5-4. The `Login` *Method*

```
Public Function Login(ByVal userName As String, ByVal password As String) As String
  If controlSocket.Connected Then

    ' Sending user name
    Dim command As String
    command = "USER " & userName & ControlChars.CrLf
    Console.WriteLine(command)
    SendCommand(command)
    GetResponse()

    ' Sending password
    command = "PASS " & password & ControlChars.CrLf
    Console.Write("PASS")  'do not display password
    SendCommand(command)
    GetResponse()
    Return replyCode
  Else
    Console.Write("Login failed because no connection is available")
  End If
  Return ""
End Function
```

The `Login` method sends the USER and PASS commands in sequence. It returns the three-digit reply code. Login is successful only if the three-digit code is 230.

SendCommand

The SendCommand method sends a command to the connected FTP server (see Listing 5-5).

Listing 5-5. The SendCommand *Method*

```
Private Sub SendCommand(ByVal command As String)
  Try
    controlSocket.Send(Encoding.ASCII.GetBytes(command), command.Length, 0)
  Catch
  End Try
End Sub
```

SendCWDCommand

The SendCWDCommand method uses the SendCommand method to send a CWD command to the connected FTP server (see Listing 5-6).

Listing 5-6. The SendCWDCommand *Method*

```
Public Sub SendCWDCommand(ByVal path As String)
  SendCommand("CWD " & path & ControlChars.CrLf)
  GetResponse()
End Sub
```

SendDELECommand

The SendDELECommand method sends a DELE command to the connected FTP server using the SendCommand method (see Listing 5-7).

Listing 5-7. The SendDELECommand *Method*

```
Public Sub SendDELECommand(ByVal filename As String)
  SendCommand("DELE " & filename & ControlChars.CrLf)
  GetResponse()
End Sub
```

SendLISTCommand

The `SendLISTCommand` method sends a LIST command to the connected FTP server and displays the returned directory list (see Listing 5-8).

Listing 5-8. The `SendLISTCommand` *Method*

```
Public Sub SendLISTCommand()

  PassiveDataConnection()
  SendCommand("LIST" & ControlChars.CrLf)
  GetResponse()

  Dim byteReceivedCount As Integer
  Dim msg As New StringBuilder(2048)
  Dim bytes(511) As Byte
  Do
    byteReceivedCount = _
      dataSocket.Receive(bytes, bytes.Length, SocketFlags.None)
    msg.Append(Encoding.ASCII.GetString(bytes, 0, byteReceivedCount))
  Loop Until byteReceivedCount = 0

  Console.WriteLine(msg.ToString())

  'because the 226 response might be sent
  'before the data connection finishes, only try to get "completion message"
  'if it's not yet sent
  If replyMessage.IndexOf("226 ") = -1 Then
    GetResponse()
  End If
End Sub
```

The `SendLISTCommand` method starts by calling the `PassiveDataConnection` and sends the LIST command:

```
PassiveDataConnection()
SendCommand("LIST" & ControlChars.CrLf)
    GetResponse()
```

The data socket instantiated in the `PassiveDataConnection` method then reads the data from the server:

```
Dim byteReceivedCount As Integer
Dim msg As New StringBuilder(2048)
Dim bytes(511) As Byte
Do
  byteReceivedCount = _
    dataSocket.Receive(bytes, bytes.Length, SocketFlags.None)
  msg.Append(Encoding.ASCII.GetString(bytes, 0, byteReceivedCount))
Loop Until byteReceivedCount = 0
```

The data then displays on the console:

```
Console.WriteLine(msg.ToString())
```

Upon sending the directory list, the server should send the 226 reply code indicating the transfer completion. However, in my testing, the 226 reply code might be sent *before* the data is received. Therefore, you call the `GetResponse` method only if the 226 reply code has not been sent:

```
If replyMessage.IndexOf("226 ") = -1 Then
  GetResponse()
End If
```

SendMKDCommand

The `SendMKCommand` method sends an MKD command to the connected FTP server (see Listing 5-9).

Listing 5-9. The `SendMKDCommand` *Method*

```
Public Sub SendMKDCommand(ByVal dir As String)
  SendCommand("MKD " & dir & ControlChars.CrLf)
  GetResponse()
End Sub
```

SendPWDCommand

The `SendPWDCommand` method sends a PWD command to the connected FTP server (see Listing 5-10).

Listing 5-10. The `SendPWDCommand` *Method*

```
Public Sub SendPWDCommand()
  SendCommand("PWD" & ControlChars.CrLf)
  GetResponse()
End Sub
```

SendRMDCommand

The `SendRMDCommand` method sends a RMD command to the connected FTP server (see Listing 5-11).

Listing 5-11. The `SendRMDCommand` *Method*

```
Public Sub SendRMDCommand(ByVal dir As String)
  SendCommand("RMD " & dir & ControlChars.CrLf)
  GetResponse()
End Sub
```

SendQUITCommand

The `SendQUITCommand` method sends a QUIT command to the connected FTP server (see Listing 5-12). After the response is received, it calls the `Shutdown` and Close `methods` of the control socket.

Listing 5-12. The `SendQUITCommand` *Method*

```
Public Sub SendQUITCommand()
  SendCommand("QUIT" & ControlChars.CrLf)
  GetResponse()
  controlSocket.Shutdown(SocketShutdown.Both)
  controlSocket.Close()
End Sub
```

SendRETRCommand

The `SendRETRCommand` method downloads a file in the connected FTP server (see Listing 5-13).

Listing 5-13. The `SendRETRCommand` *Method*

```
Public Sub SendRETRCommand(ByVal filename As String)

  Dim f As FileStream = File.Create(filename)

  SendTYPECommand("I")
  PassiveDataConnection()

  SendCommand("RETR " & filename & ControlChars.CrLf)
  GetResponse()

  Dim byteReceivedCount As Integer
  Dim totalByteReceived As Integer = 0
  Dim bytes(511) As Byte
  Do
    byteReceivedCount = _
      dataSocket.Receive(bytes, bytes.Length, SocketFlags.None)
    totalByteReceived += byteReceivedCount
    f.Write(bytes, 0, byteReceivedCount)
  Loop Until byteReceivedCount = 0

  f.Close()

  'because the 226 response might be sent
  'before the data connection finishes, only try to get "completion message"
  'if it's not yet sent
  If replyMessage.IndexOf("226 ") = -1 Then
    GetResponse()
  End If

  SendTYPECommand("A")
End Sub
```

The method starts by creating a file in the current local directory:

```
Dim f As FileStream = File.Create(filename)
```

It then changes the transmission mode to image (binary) by calling the `SendTYPECommand` method, passing `"I"` to the method:

```
SendTYPECommand("I")
```

Next, it calls the `PassiveDataConnection` method to get a data socket for the data transmission and sends a RETR command to the server:

```
PassiveDataConnection()
SendCommand("RETR " & filename & ControlChars.CrLf)
GetResponse()
```

The data socket created in the `PassiveDataConnection` method is then used to read the data stream. The incoming stream is written to the file created at the beginning of this method:

```
Dim byteReceivedCount As Integer
Dim totalByteReceived As Integer = 0
Dim bytes(511) As Byte
Do
  byteReceivedCount = _
    dataSocket.Receive(bytes, bytes.Length, SocketFlags.None)
  totalByteReceived += byteReceivedCount
  f.Write(bytes, 0, byteReceivedCount)
Loop Until byteReceivedCount = 0
```

Next, the file closes:

```
f.Close()
```

After the data transmission completes, the server closes the data connection and sends a 226 transfer completion code through the control connection. However, the 226 reply code might be sent *before* all the data is received. Therefore, you call the `GetResponse` method only if the 226 reply code has not been sent:

```
If replyMessage.IndexOf("226 ") = -1 Then
  GetResponse()
End If
```

Finally, it changes the mode back to ASCII:

```
SendTYPECommand("A")
```

SendSTORCommand

The `SendSTORCommand` method uploads a file in the connected FTP server (see Listing 5-14).

Listing 5-14. The `SendSTORCommand` *Method*

```
Public Sub SendSTORCommand(ByVal filename As String)

  Dim f As FileStream = File.Open(filename, FileMode.Open)
  SendTYPECommand("I")
  PassiveDataConnection()

  SendCommand("STOR " & filename & ControlChars.CrLf)
  GetResponse()

  Dim byteReadCount As Integer
  Dim totalByteSent As Integer
  Dim bytes(511) As Byte

  Do
    byteReadCount = f.Read(bytes, 0, bytes.Length)
    If byteReadCount <> 0 Then
      dataSocket.Send(bytes, byteReadCount, SocketFlags.None)
      totalByteSent += byteReadCount
    End If
  Loop Until byteReadCount = 0

  dataSocket.Shutdown(SocketShutdown.Both)
  dataSocket.Close()

  f.Close()
  GetResponse()

  SendTYPECommand("A")
End Sub
```

The method starts by opening the file to upload to the connected FTP server:

```
Dim f As FileStream = File.Open(filename, FileMode.Open)
```

It then changes the transmission mode to image (binary) and calls the `PassiveDataConnection` to obtain a data socket for the file transmission:

```
SendTYPECommand("I")
PassiveDataConnection()
```

Then it sends a STOR command to indicate to the server that it is going to send a file to that server:

```
SendCommand("STOR " & filename & ControlChars.CrLf)
GetResponse()
```

The data socket created by the `PassiveDataConnection` transfers the file.

```
Dim byteReadCount As Integer
Dim totalByteSent As Integer
Dim bytes(511) As Byte

Do
  byteReadCount = f.Read(bytes, 0, bytes.Length)
  If byteReadCount <> 0 Then
    dataSocket.Send(bytes, byteReadCount, SocketFlags.None)
    totalByteSent += byteReadCount
  End If
Loop Until byteReadCount = 0
```

After the file transfer completes, the data socket closes:

```
dataSocket.Shutdown(SocketShutdown.Both)
dataSocket.Close()
```

Then, the file closes and the mode switches back to ASCII:

```
f.Close()
GetResponse()
SendTYPECommand("A")
```

SendSYSTCommand

The `SendSYSTCommand` method sends a SYST method to the connected FTP server (see Listing 5-15).

Listing 5-15. The `SendSYSTCommand` *Method*

```
Public Sub SendSYSTCommand()
  SendCommand("SYST" & ControlChars.CrLf)
  GetResponse()
End Sub
```

SendTYPECommand

You use the `SendTYPECommand` method to change the transmission mode from the client to the server (see Listing 5-16).

Listing 5-16. The `SendTYPECommand` *Method*

```
Public Sub SendTYPECommand(ByVal type As String)
  SendCommand("TYPE " & type & ControlChars.CrLf)
  GetResponse()
End Sub
```

Main

The `Main` static method is the entry point of the program. It does the following:

- Ensures that the program is invoked using the correct number of arguments.
- Controls the program flow with a `While` loop so that the user can enter one FTP command after the execution of another.
- Invokes the correct method upon receiving a valid user input.
- Displays an error message on receiving an invalid user input.

Listing 5-17 shows the `Main` method.

Listing 5-17. The `Main` *Method*

```
Public Shared Sub Main(ByVal args As String())
  If args.Length <> 3 Then
    Console.WriteLine("usage: NETFTP server username password")
  Else
    Dim ftp As New NETFTP()
```

```
    ftp.Connect(args(0))
    Dim replyCode As String = ftp.Login(args(1), args(2))

    If replyCode.Equals("230") Then
      'login successful, allow user to type in commands
      ftp.SendSYSTCommand()
      ftp.SendTYPECommand("A")

      Dim command As String = ""
      Try
        While Not command.ToUpper.Equals("QUIT")
          Console.Write("NETFTP>")
          command = Console.ReadLine().Trim()
          If command.ToUpper.Equals("PWD") Then
            ftp.SendPWDCommand()
          ElseIf command.ToUpper.StartsWith("CWD") Then
            If command.Length > 3 Then
              Dim path As String = command.Substring(4).Trim()
              If path.Equals("") Then
                Console.WriteLine("Please specify the directory to change to")
              Else
                ftp.SendCWDCommand(path)
              End If
            End If
          ElseIf command.ToUpper.StartsWith("DELE") Then
            If command.Length > 4 Then
              Dim path As String = command.Substring(5).Trim()
              If path.Equals("") Then
                Console.WriteLine("Please specify the file to delete")
              Else
                ftp.SendDELECommand(path)
              End If
            End If
          ElseIf command.ToUpper.Equals("LIST") Then
            ftp.SendLISTCommand()
          ElseIf command.ToUpper.StartsWith("MKD") Then
            If command.Length > 3 Then
              Dim dir As String = command.Substring(4).Trim()
                If dir.Equals("") Then
                  Console.WriteLine("Please specify the name↵
of the directory to create")
              Else
                ftp.SendMKDCommand(dir)
              End If
            End If
```

```
            ElseIf command.ToUpper.Equals("QUIT") Then
            ElseIf command.ToUpper.StartsWith("RMD") Then
              If command.Length > 3 Then
                Dim dir As String = command.Substring(4).Trim()
                If dir.Equals("") Then
                  Console.WriteLine("Please specify the name↵
of the directory to delete")
                Else
                  ftp.SendRMDCommand(dir)
                End If
              End If
            ElseIf command.ToUpper.StartsWith("RETR") Then
              If command.Length > 4 Then
                Dim filename As String = command.Substring(5).Trim()
                If filename.Equals("") Then
                  Console.WriteLine("Please specify a file to retrieve")
                Else
                  ftp.SendRETRCommand(filename)
                End If
              End If
            ElseIf command.ToUpper.StartsWith("STOR") Then
              If command.Length > 4 Then
                Dim filename As String = command.Substring(5).Trim()
                If filename.Equals("") Then
                  Console.WriteLine("Please specify a file to store")
                Else
                  ftp.SendSTORCommand(filename)
                End If
              End If
            Else
              Console.WriteLine("Invalid command.")
            End If
          End While
          ftp.SendQUITCommand()
        Catch e As Exception
          Console.WriteLine(e.ToString())
        End Try
        Console.WriteLine("Thank you for using NETFTP.")
      Else
        ftp.SendQUITCommand()
        Console.WriteLine("Login failed. Please try again.")
      End If
    End If
  End Sub
```

The `Main` method starts by checking that the user passes three arguments: server, username, and password. If the number of arguments is not three, it prints the following message on the console:

```
usage: NETFTP server username password
```

If the number of arguments is correct, the `Main` method instantiates a NETFTP object and uses the first argument (server) to connect to the remote server by calling the `Connect` method:

```
ftp.Connect(args(0))
```

Once connected, the `Main` method uses the second and third argument to log in by calling the `Login` method:

```
Dim replyCode As String = ftp.Login(args(1), args(2))
```

The remote server returning a reply code of 230 indicates a successful login. If login was successful, it calls the `SendSYSTCommand` and `SendTYPECommand` methods. The `SendSYSTCommand` method prints the server's operating system information, and the `SendTYPECommand` method is called by passing `"A"` as the argument. This in effect changes the transfer mode to ASCII:

```
If replyCode.Equals("230") Then
  'login successful, allow user to type in commands
  ftp.SendSYSTCommand()
  ftp.SendTYPECommand("A")
```

After the user is successfully logged in, the program enters a `While` loop. Inside the `While` loop is the code that gets the user's input and sends the corresponding FTP command. Control exits the `While` loop when the user types *QUIT*:

```
Dim command As String = ""
Try
  While Not command.ToUpper.Equals("QUIT")
```

Inside the `While` loop, the `Main` method first displays the `NETFTP>` prompt and uses the `Console` class's `ReadLine` to read the user input:

```
Console.Write("NETFTP>")
command = Console.ReadLine().Trim()
```

Then, it goes through a series of `If` commands to execute code based on the command entered by the user. Valid commands are PWD, CWD, DELE, LIST, MKD, QUIT, RMD, RETR, and STOR.

If the command equals PWD, the `SendPWDCommand` method is called:

```
If command.ToUpper.Equals("PWD") Then
  ftp.SendPWDCommand()
```

If the command is CWD, it ensures that there is a parameter, a path, after the CWD command and that the parameter does not consist of spaces only. If the path looks valid, it calls the `SendCWDCommand` method. Otherwise, it prints a warning on the console:

```
ElseIf command.ToUpper.StartsWith("CWD") Then
  If command.Length > 3 Then
    Dim path As String = command.Substring(4).Trim()
    If path.Equals("") Then
      Console.WriteLine("Please specify the directory to change to")
    Else
      ftp.SendCWDCommand(path)
    End If
  End If
```

If the command is DELE, it makes sure there is a valid parameter, a filename. If the command has a valid parameter, it invokes the `SendDELECommand` method. Otherwise, it prints a warning to the user:

```
ElseIf command.ToUpper.StartsWith("DELE") Then
  If command.Length > 4 Then
    Dim path As String = command.Substring(5).Trim()
    If path.Equals("") Then
      Console.WriteLine("Please specify the file to delete")
    Else
      ftp.SendDELECommand(path)
    End If
  End If
```

If the command equals LIST, it invokes the `SendLISTCommand` method:

```
ElseIf command.ToUpper.Equals("LIST") Then
  ftp.SendLISTCommand()
```

If the command starts with MKD and there is a valid parameter (a directory name), it calls the `SendMKDCommand` method. Otherwise, a warning displays on the console:

```
ElseIf command.ToUpper.StartsWith("MKD") Then
  If command.Length > 3 Then
    Dim dir As String = command.Substring(4).Trim()
    If dir.Equals("") Then
      Console.WriteLine("Please specify the name of the directory to create")
    Else
      ftp.SendMKDCommand(dir)
    End If
  End If
```

If the command equals `QUIT`, it does not do anything as this will be captured by the conditional statement in the `While` loop:

```
ElseIf command.ToUpper.Equals("QUIT") Then
```

If the command starts with RMD and there is a valid parameter (a directory name to remove), it invokes the `SendRMDCommand` method. Otherwise, a warning displays on the console:

```
ElseIf command.ToUpper.StartsWith("RMD") Then
  If command.Length > 3 Then
    Dim dir As String = command.Substring(4).Trim()
    If dir.Equals("") Then
      Console.WriteLine("Please specify the name of the directory to delete")
    Else
      ftp.SendRMDCommand(dir)
    End If
  End If
```

If the command starts with RETR and there is a parameter (a filename), the `SendRETRCommand` method is invoked. Otherwise, it prints a warning on the console:

```
ElseIf command.ToUpper.StartsWith("RETR") Then
  If command.Length > 4 Then
    Dim filename As String = command.Substring(5).Trim()
    If filename.Equals("") Then
      Console.WriteLine("Please specify a file to retrieve")
    Else
      ftp.SendRETRCommand(filename)
    End If
  End If
```

If the command starts with `STOR` and there is a parameter (a filename), it invokes the `SendSTORCommand` method. Otherwise, it prints a warning on the console:

```
ElseIf command.ToUpper.StartsWith("STOR") Then
  If command.Length > 4 Then
    Dim filename As String = command.Substring(5).Trim()
    If filename.Equals("") Then
      Console.WriteLine("Please specify a file to store")
    Else
      ftp.SendSTORCommand(filename)
    End If
  End If
```

Finally, if the command is none of the valid commands, it simply prints "Invalid command" on the console:

```
Else
  Console.WriteLine("Invalid command.")
End If
```

When the user types QUIT, the control exits the `While` loop, and the `SendQUITCommand` method is invoked:

```
ftp.SendQUITCommand()
```

Before closing itself, the program prints a thank you message.

```
Console.WriteLine("Thank you for using NETFTP.")
```

Implementing the Project

The project for this chapter is an FTP client application with a graphical user interface, as shown in Figure 5-2. This section starts with a general overview of the application. It then describes each class in detail.

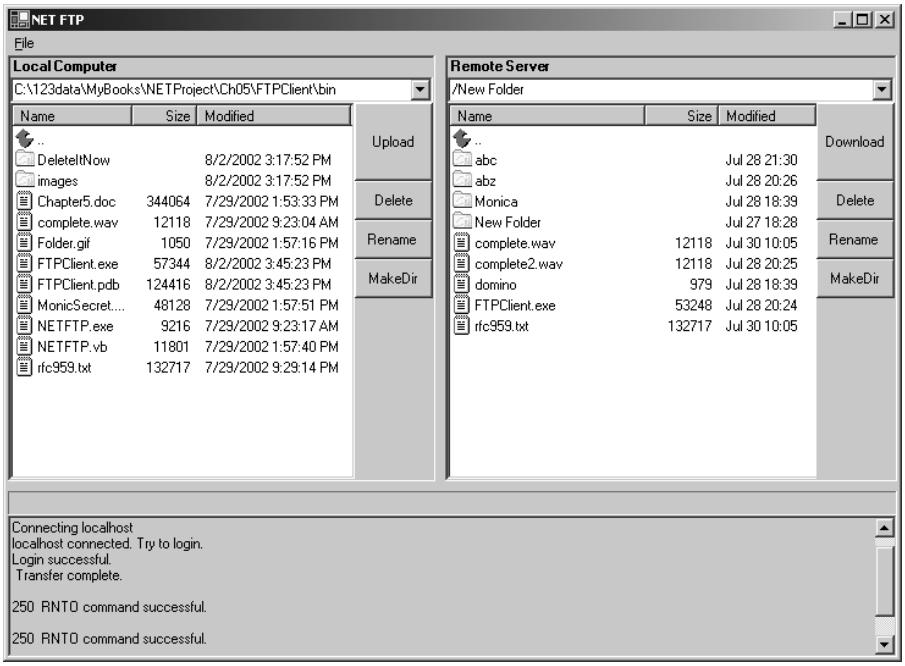

Figure 5-2. The FTP client application

To get a feel for the application, you are encouraged to try this application by double-clicking the `Form1.exe` file in the `listings/Ch05/Project` directory.

When the application activates, it retrieves the content of the local current directory and displays it on the left panel of the form.

Before you can do a file transfer, you need to connect to an FTP server. You can connect and log in at the same time by pressing F3 or selecting File ➤ Connect. To log in, you type your login details in the Login Form window, as shown in Figure 5-3.

Figure 5-3. The Login Form window

You need to enter the server, the username, and the password into the Login Form window and then click the OK button.

If the connection is successful, the content of the remote home directory displays in the right panel of the form. If the login fails, the Login Form window remains open until you enter the correct login details or until you click the Cancel button.

The four buttons to the right of the left panel are for manipulating local files and directories, and the buttons to the right of the right panel are for manipulating remote files and directories. You can change the local or remote directory by double-clicking the directory icon on both panels.

To upload a file, you can select a file from the local computer and click the Upload button. Alternatively, you can simply double-click the file icon on the left panel.

To download a file, select a file from the remote computer and then click the Download button. Or, you can double-click the file icon.

Creating the Class Diagram

Figure 5-4 shows the class diagram for this application.

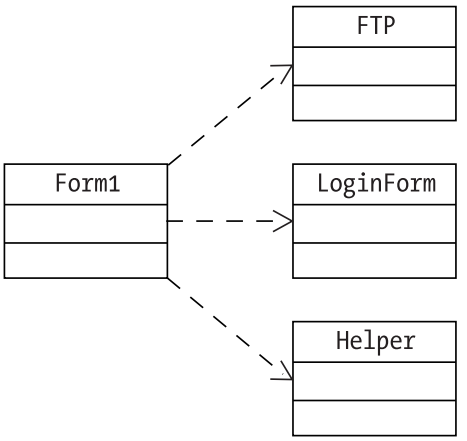

Figure 5-4. The class diagram

The application is comprised of the following classes:

- **FTP**: Contains properties and methods to send FTP commands to an FTP server.
- **Form1**: The main form of the application.
- **Helper**: Contains static methods used by the `Form1` class
- **LoginForm**: Represents a form for the user to log in.
- **Three EventArgs subclasses**: `EndDownloadEventArgs`, `EndUploadEventArgs`, and `TransferProgressChangedEventArgs`. These are event argument classes used in several delegates in the FTP class.

You will now learn about the classes starting with the easiest.

Creating the Helper Class

You can find the `Helper` class in the `Helper.vb` file under the project's directory. It provides two static methods used by the `Form1` class: `IsDirectory` and `IsDirectoryItem`.

The Helper Class's Methods

The following sections describe the two methods of the `Helper` class.

IsDirectory

The `IsDirectory` method accepts a string argument and returns `True` if the specified string is a path to a directory.

The method definition is as follows:

```
Public Shared Function IsDirectory(ByVal path As String) As Boolean
  If File.Exists(path) Or Directory.Exists(path) Then
    ' it is a file or a directory
    Dim attr As FileAttributes = File.GetAttributes(path)
    If (attr And FileAttributes.Directory) = FileAttributes.Directory Then
      Return True
    End If
  End If
End Function
```

IsDirectoryItem

The `IsDirectoryItem` method accepts a `System.Windows.Form.ListViewItem` and returns `True` if the item's `ImageIndex` is 1, the image index of the folder icon. Its definition is as follows:

```
Public Shared Function IsDirectoryItem(ByVal item As ListViewItem) As Boolean
  If item.ImageIndex = 1 Then
    Return True
  Else
    Return False
  End If
End Function
```

Creating the LoginForm Class

You can find the `LoginForm` class in the `LoginForm.vb` file in the project's directory. It represents the login form for the user to login. This form displays as a modal dialog box from the `Form1` class when the user attempts to connect to a remote server.

The class contains three `Label` controls (`label1`, `label2`, and `label3`), three `TextBox` controls (`serverTextBox`, `userTextBox`, and `passwordTextBox`), and two `Button` controls (`okButton` and `cnlButton`).

The `passwordTextBox` control accepts the user's password, and the characters entered are masked by setting its `PasswordChar` property as follows:

```
Me.passwordTextBox.PasswordChar = Microsoft.VisualBasic.ChrW(42)
```

You set the `okButton` control's `DialogResult` property to `System.Windows.Forms.DialogResult.OK` so that when the form is shown as a modal dialog box, it returns `DialogResult.OK` when the `okButton` control is clicked:

```
Me.okButton.DialogResult = System.Windows.Forms.DialogResult.OK
```

On the other hand, you set the `cnlButton.DialogResult` property to `System.Windows.Forms.DialogResult.Cancel`. When the form displays as a modal dialog box, clicking the `cnlButton` control results in the form returning `Dialog.Cancel`:

```
Me.cnlButton.DialogResult = System.Windows.Forms.DialogResult.Cancel
```

The okButton control's Click event is wired with the okButton_Click event handler. This event handler populates three private fields with the values entered by the user into the serverTextBox, userTextBox, and passwordTextBox controls:

```
Private Sub okButton_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles okButton.Click
  userNameField = userTextBox.Text
  passwordField = passwordTextBox.Text
  serverField = serverTextBox.Text
  Me.Close()
End Sub
```

The last line of this event handler also closes the form.

After the form returns—in other words, when either okButton or cnlButton is clicked—you can obtain the values of serverField, userNameField, and passwordField from the three read-only properties: Server, UserName, and Password:

```
Public ReadOnly Property Server() As String
  Get
    Return serverField
  End Get
End Property

Public ReadOnly Property UserName() As String
  Get
    Return userNameField
  End Get
End Property

Public ReadOnly Property Password() As String
  Get
    Return passwordField
  End Get
End Property
```

Creating the FTP Class

You can find the FTP class in the FTP.vb file in the project's directory. This class encapsulates functions to communicate with an FTP server. Using this class, you can connect to a remote FTP server, log in, print the working directory, change the directory, delete and rename a remote file, and download and upload a file.

Some of the functions in the FTP class are similar to those in the NETFTP class discussed previously. However, you use a separate thread for downloading and uploading a file to improve the perceived performance. For synchronization, a Boolean called transferring prevents multiple upload/download at the same time. The Upload and Download methods return immediately if the value of transferring is True.

Finally, the FTP class has five public events as described in "The FTP Class's Events" section.

The FTP Class's Declaration

The FTP class contains the following variable declaration:

```
Private port As Integer = 21
Private controlSocket, dataSocket As Socket
Private serverAddress As String
Private directoryListField As String
'the thread used for uploading and downloading files
Private dataTransferThread As Thread
'indicates whether dataTransferThread is being used
'if it is, do not allow another operation
Private transferring As Boolean

'for transferring filename and localDir when calling DoUpload and
'DoDownload
Private filename, localDir As String

Public replyMessage As String
Public replyCode As String
```

The FTP Class's Properties

The FTP class has two read-only properties.

Connected

The Connected property indicates whether the control socket is connected:

```
Public ReadOnly Property Connected() As Boolean
  Get
    If Not controlSocket Is Nothing Then
```

```
      Return controlSocket.Connected
    Else
      Return False
    End If
  End Get
End Property
```

DirectoryList

The `DirectoryList` property returns the directory list obtained from the `GetDirList` method in raw form:

```
Public ReadOnly Property DirectoryList() As String
  Get
    Return directoryListField
  End Get
End Property
```

The FTP Class's Methods

The following sections describe the methods in the FTP class.

ChangeDir

The `ChangeDir` method changes the current remote directory by sending the CWD command:

```
Public Sub ChangeDir(ByVal path As String)
  SendCommand("CWD " & path & ControlChars.CrLf)
  GetResponse()
End Sub
```

ChangeToAsciiMode

The `ChangeToAsciiMode` method changes the transfer mode to ASCII:

```
Public Sub ChangeToAsciiMode()
  SendTYPECommand("A")
End Sub
```

Connect

This method connects to a remote server:

```
Public Sub Connect(ByVal server As String)
  Try
    controlSocket = New _
      Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp)
    controlSocket.Connect(New _
      IPEndPoint(Dns.Resolve(server).AddressList(0), port))
  Catch e As Exception
    Console.WriteLine(e.ToString())
    Return
  End Try
  If controlSocket.Connected Then
    Console.WriteLine("Connected. Waiting for reply...")
    GetResponse()
  Else
    Console.WriteLine("Couldn't connect.")
  End If
End Sub
```

DeleteDir

The `DeleteDir` method deletes a directory on the connected server by sending an RMD command. A 2xx reply code indicates a successful RMD command. The method definition is as follows:

```
Public Function DeleteDir(ByVal dir As String) As Boolean
  SendCommand("RMD " & dir & ControlChars.CrLf)
  GetResponse()
  If replyCode.StartsWith("2") Then
    Return True
  Else
    Return False
  End If
End Function
```

DeleteFile

The `DeleteFile` method deletes a file on the connected server by sending a DELE command. A 2xx reply code indicates a successful DELE command. The method definition is as follows:

```
Public Function DeleteFile(ByVal filename As String) As Boolean
  SendCommand("DELE " & filename & ControlChars.CrLf)
  GetResponse()
  If replyCode.StartsWith("2") Then
    Return True
  Else
    Return False
  End If
End Function
```

Disconnect

The `Disconnect` method disconnects from the remote server:

```
Public Sub Disconnect()
  If controlSocket.Connected Then
    SendCommand("QUIT" & ControlChars.CrLf)
    GetResponse()
    controlSocket.Shutdown(SocketShutdown.Both)
    controlSocket.Close()
  End If
End Sub
```

DoDownload

The `DoDownload` method does the actual file download. This method is called from the `Download` method. The following is the `DoDownload` method:

```
Public Sub DoDownload()
  OnBeginDownload(New EventArgs())
  Dim completePath As String = Path.Combine(localDir, filename)
  Try

    Dim f As FileStream = File.Create(completePath)
    SendTYPECommand("I")
    PassiveDataConnection()
    SendCommand("RETR " & filename & ControlChars.CrLf)
    GetResponse()
    Dim byteReceivedCount As Integer
    Dim totalByteReceived As Integer = 0
    Dim bytes(511) As Byte
    Do
      byteReceivedCount = _
```

```
            dataSocket.Receive(bytes, bytes.Length, SocketFlags.None)
          totalByteReceived += byteReceivedCount

          f.Write(bytes, 0, byteReceivedCount)
          OnTransferProgressChanged(New _
            TransferProgressChangedEventArgs(totalByteReceived))
        Loop Until byteReceivedCount = 0

        f.Close()
        'because the 226 response might be sent
        'before the data connection finishes, only try to get "completion message"
        'if it's not yet sent
        If replyMessage.IndexOf("226 ") = -1 Then
          GetResponse()
        End If

        SendTYPECommand("A")
      Catch
      End Try
      Dim e As New EndDownloadEventArgs()
      e.Message = "Finished downloading " & filename
      OnEndDownload(e)
      transferring = False
    End Sub
```

The `DoDownload` method starts by raising the `BeginDownload` event:

```
OnBeginDownload(New EventArgs())
```

The `DoDownload` method is the method assigned to a new thread created in the `Download` method. Prior to starting the new thread, the `Download` method sets the `localDir` and filename variables. The `localDir` is the current local directory to which the downloaded file will be saved. The filename variable contains the name of the file to be downloaded.

The next thing the `DoDownload` method does after raising the `BeginDownload` event is combine `localDir` and `filename`:

```
Dim completePath As String = Path.Combine(localDir, filename)
```

Next, it creates a file on the local machine using the combined string of `localDir` and `filename`:

```
Dim f As FileStream = File.Create(completePath)
```

Then, it changes the transfer mode to image (binary) and calls the `PassiveDataConnection` method. The latter constructs a data socket to be used for the file transfer:

```
SendTYPECommand("I")
PassiveDataConnection()
```

The actual file download starts when a RETR command is sent:

```
SendCommand("RETR " & filename & ControlChars.CrLf)
GetResponse()
```

Then, the data socket resulted from the `PassiveDataConnection` method receives the data stream:

```
Dim byteReceivedCount As Integer
Dim totalByteReceived As Integer = 0
Dim bytes(511) As Byte
Do
  byteReceivedCount = _
    dataSocket.Receive(bytes, bytes.Length, SocketFlags.None)
  totalByteReceived += byteReceivedCount

  f.Write(bytes, 0, byteReceivedCount)
  OnTransferProgressChanged(New _
    TransferProgressChangedEventArgs(totalByteReceived))
Loop Until byteReceivedCount = 0
```

Note from the previous `Do` loop that the `TransferProgressChanged` event raises after each invocation of the data socket's `Received` method, passing the total number of bytes received so far. The user of the FTP class can use this event to notify the user of the progress of the file transfer, for example, by using a progress bar.

Afterward, the file closes:

```
f.Close()
```

After the file transfer completes, the server sends the 226 reply code. However, sometimes this reply code is received even before the whole data transferred is received. Therefore, you check to see that a 226 reply code has not been received prior to calling the `GetResponse` method:

```
If replyMessage.IndexOf("226 ") = -1 Then
  GetResponse()
End If
```

It then changes the mode to ASCII:

```
  SendTYPECommand("A")
Catch
End Try
```

Finally, the `EndDownload` event triggers, passing the "Finished downloading *filename*" message, where *filename* is the name of the file downloaded and the transferring `Boolean` resets to allow a future file transfer:

```
Dim e As New EndDownloadEventArgs()
e.Message = "Finished downloading " & filename
OnEndDownload(e)
transferring = False
```

DoUpload

The `DoUpload` method does the actual file upload. This method is called from the `Upload` method. The `DoUpload` method starts by raising the `BeginUpload` event:

```
OnBeginUpload(New EventArgs())
```

The `DoUpload` method is the method assigned to a new thread created in the `Upload` method. Prior to starting the new thread, the `Upload` method sets the `localDir` and `filename` variables. The `localDir` is the current local directory to which the downloaded file will be saved. The `filename` variable contains the name of the file to be downloaded.

The next thing the `DoUpload` method does after raising the `BeginUpload` event is combine `localDir` and `filename`:

```
Dim completePath As String = Path.Combine(localDir, filename)
```

Then it opens the file to upload, changes the mode to ASCII, and calls the `PassiveDataConnection` method. The `PassiveDataConnection` method constructs a data socket to be used for the file transfer:

```
Dim f As FileStream = _
  File.Open(completePath, FileMode.Open, FileAccess.Read)
SendTYPECommand("I")
PassiveDataConnection()
```

The actual file upload starts when a STOR command is sent:

```
SendCommand("STOR " & filename & ControlChars.CrLf)
GetResponse()
```

Then, the data socket resulted from the `PassiveDataConnection` method receives the data stream:

```
Dim byteReadCount As Integer
Dim totalByteSent As Integer
Dim bytes(511) As Byte

Do
  byteReadCount = f.Read(bytes, 0, bytes.Length)
  If byteReadCount <> 0 Then
    dataSocket.Send(bytes, byteReadCount, SocketFlags.None)
    totalByteSent += byteReadCount
    OnTransferProgressChanged( _
      New TransferProgressChangedEventArgs(totalByteSent))
  End If
Loop Until byteReadCount = 0
```

Note from the previous `Do` loop that the `TransferProgressChanged` event raises after each invocation of the data socket's `Send` method, passing the total number of bytes sent so far. The user of the `FTP` class can use this event to notify the user of the progress of the file transfer, for example, by using a progress bar.

Afterward, the data socket and the file close:

```
dataSocket.Shutdown(SocketShutdown.Both)
dataSocket.Close()
f.Close()
```

When the data socket closes, the server knows that the file transfer is completed and sends a reply code that you receive using the `GetResponse` method:

```
GetResponse()
```

Then, it changes the mode back to ASCII:

```
SendTYPECommand("A")
```

Finally, the `EndUpload` event triggers, passing the "Finished uploading *filename*" message, where *filename* is the name of the file uploaded and the transferring `Boolean` resets to allow a future file transfer:

```
Dim ev As New EndUploadEventArgs()
ev.Message = "Finished uploading " & filename
OnEndUpload(ev)
transferring = False
```

Download

The `Download` method checks if file transfer is allowed and, if it is, creates a new thread to download a file:

```
Public Sub Download(ByVal filename As String, ByVal localdir As String)
  If Not transferring Then
    transferring = True
    Me.filename = filename
    Me.localDir = localdir
    dataTransferThread = _
      New Thread(New ThreadStart(AddressOf DoDownload))
    dataTransferThread.Start()
  End If
End Sub
```

GetRemoteDirectory

`GetRemoteDirectory` is a helper method used to obtain the directory name at the remote server. The argument passed to this function is a string that is the server reply after a PWD command is sent. Therefore, the argument has the following format:

```
"path" is current directory
```

This method returns the "*path*" portion of the argument. The definition for this method is as follows:

```
Private Function GetRemoteDirectory(ByVal message As String) As String
  'message is the server response upon sending the "PWD" command
  'its format is something like: "path" is current directory
  'this function obtains the string between the double quotes
  Dim path As String = ""
  Dim index As Integer = message.IndexOf("""")
  If index <> -1 Then
    Dim index2 As Integer = message.IndexOf("""", index + 1)
    If index2 <> -1 Then
      path = message.Substring(index + 1, index2 - index - 1)
    End If
  End If
  Return path
End Function
```

GetResponse

The `GetResponse` method receives the server reply and is the same as the `GetResponse` method in the `NETFTP` class.

Login

The `Login` method logs in to a connected FTP server and is similar to the `Login` method in the `NETFTP` class. The difference is that this method returns a `Boolean` because the server's reply is tested at the end of the method, as follows:

```
If replyCode.Equals("230") Then
  Return True
Else
  Return False
End If
```

MakeDir

The `MakeDir` method creates a new directory in the remote server by sending an MKD command:

```
Public Sub MakeDir(ByVal dir As String)
  SendCommand("MKD " & dir & ControlChars.CrLf)
  GetResponse()
End Sub
```

PassiveDataConnection

The `PassiveDataConnection` method is the same as the `PassiveDataConnection` method in the `NETFTP` class.

Rename

The `Rename` method changes the name of a remote file. It does so by sending the RNFR command and the RNTO command in sequence. Its definition is as follows:

```
Public Sub Rename(ByVal renameFrom As String, ByVal renameTo As String)
  SendCommand("RNFR " & renameFrom & ControlChars.CrLf)
  GetResponse()
  Console.WriteLine(replyCode & " " & replyMessage)
  SendCommand("RNTO " & renameTo & ControlChars.CrLf)
  GetResponse()
  Console.WriteLine(replyCode & " " & replyMessage)
End Sub
```

SendCommand

The `SendCommand` method sends a specified command to the connected server. Its definition is as follows:

```
Private Sub SendCommand(ByVal command As String)
  Try
    controlSocket.Send(Encoding.ASCII.GetBytes(command), command.Length, 0)
  Catch
  End Try
End Sub
```

SendTYPECommand

The `SendTYPECommand` method is the same as the `SendTYPECommand` method in the `NETFTP` class.

Upload

The `Upload` method checks if file transfer is allowed by checking the value of transferring. If the value is `False`, file transfer is allowed. It then creates a new thread for the file transfer and starts the `DoUpload` method:

```
Public Sub Upload(ByVal filename As String, ByVal localDir As String)
  If Not transferring Then
    transferring = True
    Me.filename = filename
    Me.localDir = localDir
    dataTransferThread = New Thread(New ThreadStart(AddressOf DoUpload))
    dataTransferThread.Start()
  End If
End Sub
```

The FTP Class's Events

The `FTP` class can raise the following events: `BeginDownload`, `EndDownload`, `BeginUpload`, `EndUpload`, and `TransferProgressChanged`. The first four are self-explanatory. The `TransferProgressChanged` event raises several times during the download and upload processes. The user of the FTP class can capture this event to obtain the number of bytes of data transfer so far.

The definitions of public delegates are as follows:

```
Public Delegate Sub BeginDownloadEventHandler(ByVal sender As Object, _
  ByVal e As EventArgs)

Public Delegate Sub EndDownloadEventHandler(ByVal sender As Object, _
  ByVal e As EndDownloadEventArgs)

Public Delegate Sub BeginUploadEventHandler(ByVal sender As Object, _
  ByVal e As EventArgs)

Public Delegate Sub EndUploadEventHandler(ByVal sender As Object, _
  ByVal e As EndUploadEventArgs)

Public Delegate Sub TransferProgressChangedEventHandler(ByVal sender As Object, _
  ByVal e As TransferProgressChangedEventArgs)
```

Creating the EventArgs Subclasses

The following are the three subclasses of `EventArgs` class:

```
Public Class EndDownloadEventArgs : Inherits EventArgs
  Public Message As String
End Class

Public Class EndUploadEventArgs : Inherits EventArgs
  Public Message As String
End Class

Public Class TransferProgressChangedEventArgs : Inherits EventArgs
  Public TransferredByteCount As Integer

  Public Sub New()
  End Sub

  Public Sub New(ByVal size As Integer)
    TransferredByteCount = size
  End Sub
End Class
```

Creating the Form1 Class

You can find the `Form1` class in the `Form1.vb` file in the project's directory. It represents the main form in the application. This section starts by showing various controls used in the form. It then describes how those controls connect together. The description makes frequent references to the class's members, each of which is given in detail at the end of the section.

To understand how the form works, let's start with its visual description. Figure 5-5 shows various controls on Form1.

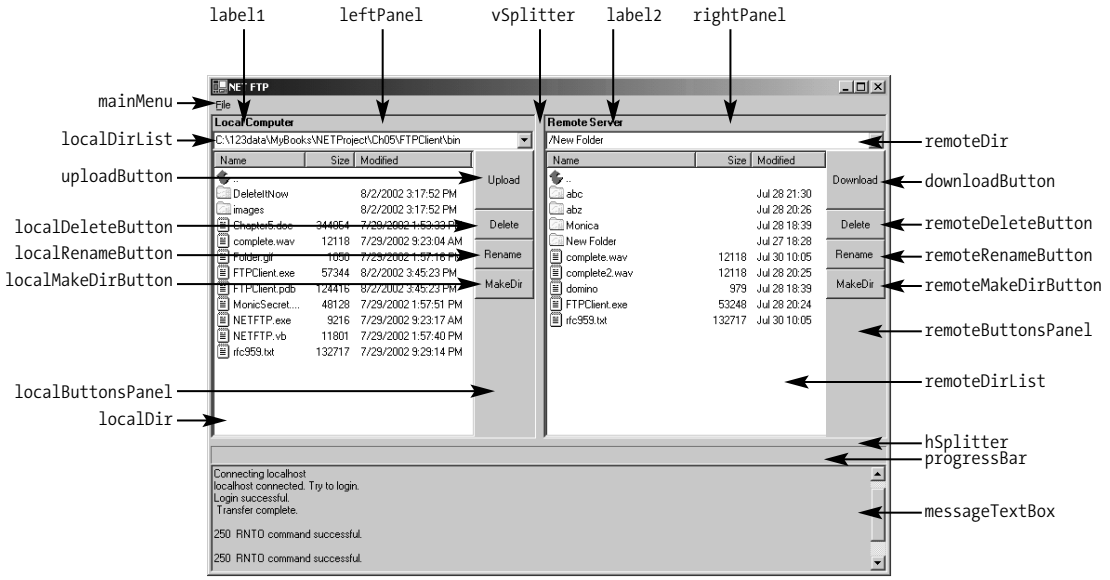

Figure 5-5. Control names on `Form1`

You can find the declaration of the controls in the `Form1` class body:

```
Private hSplitter As System.Windows.Forms.Splitter
Private vSplitter As System.Windows.Forms.Splitter
Private leftPanel As System.Windows.Forms.Panel
Private rightPanel As System.Windows.Forms.Panel
Private localButtonsPanel As System.Windows.Forms.Panel
Private remoteButtonsPanel As System.Windows.Forms.Panel
Private progressBar As System.Windows.Forms.ProgressBar
Private label1 As System.Windows.Forms.Label
Private label2 As System.Windows.Forms.Label
Private localDir As System.Windows.Forms.ComboBox
Private localDeleteButton As System.Windows.Forms.Button
Private localRenameButton As System.Windows.Forms.Button
Private localMakeDirButton As System.Windows.Forms.Button
Private localDirList As System.Windows.Forms.ListView
Private uploadButton As System.Windows.Forms.Button
Private remoteDir As System.Windows.Forms.ComboBox
Private remoteDirList As System.Windows.Forms.ListView
Private remoteDeleteButton As System.Windows.Forms.Button
Private downloadButton As System.Windows.Forms.Button
Private remoteRenameButton As System.Windows.Forms.Button
```

```
Private remoteMakeDirButton As System.Windows.Forms.Button
Private messageTextBox As System.Windows.Forms.TextBox
Private mainMenu As System.Windows.Forms.MainMenu
Private fileMenuItem As System.Windows.Forms.MenuItem
Private connectFileMenuItem As System.Windows.Forms.MenuItem
Private exitFileMenuItem As System.Windows.Forms.MenuItem
Private imageList As System.Windows.Forms.ImageList
```

When first instantiated, the class's constructor calls the `InitializeComponent` method that instantiated the controls used in the form. At the last line, the `InitializeComponent` method calls the `InitializeControls` method, which wires events with event handlers, loads images, and so on.

The form has an `ImageList` control with three images used to represent a parent directory, a folder, and a file. The image files (`Up.gif`, `Folder.gif`, and `File.gif`) are located in the `images` directory under the project's directory. You add the images to `imageList` in the `InitializeControls` method:

```
imageList.Images.Add(Bitmap.FromFile("./images/Up.gif"))
imageList.Images.Add(Bitmap.FromFile("./images/Folder.gif"))
imageList.Images.Add(Bitmap.FromFile("./images/File.gif"))
```

At the end of its body, the `InitializeControls` method calls the following two methods:

```
SelectLocalDirectory(localCurrentDir)
Log("Welcome. Press F3 for quick login.")
```

The `SelectLocalDirectory` method populates the `localDirList` control with the list of subdirectories/files in the current directory, and the `Log` method displays a message in the `messageTextBox` control. The current local directory displays in the `localDir ComboBox` control.

Without being connected to a remote server, you can browse through your local directory by double-clicking the parent directory icon and any subdirectory in the current directory.

You can even do some simple file/directory manipulations such as the following:

- Create a new directory by clicking the `localMakeDirButton` control. The `Click` event of the `localMakeDirButton` control is handled by `localMakeDirButton_Click`, which calls the `MakeLocalDir` method.

- Delete a file/subdirectory by selecting a file/directory in the `localDirList` control and clicking the `localDeleteButton` control. The `Click` event of the `localDirList` control is wired with the `localDeleteButton_Click` event handler. This event handler calls the `DeleteLocalFile` method.

- Rename a file/directory by selecting a file/directory in the `localDirList` control and clicking the `localRenameButton` control. This button's `Click` event is wired to the `localRenameButton_Click` event handler, which calls the `RenameLocalFile` method.

However, using an FTP client application, you will want to connect to a remote server. You do this by pressing F3 or by selecting File ➤ Connect. Both the F3 shortcut and the Connect menu item activates the `Connect` method. The `Connect` method has two functions. When no FTP server is connected, it connects to the server. When there is an FTP server connected, it disconnects the connection.

Connecting to a remote server requires you to enter the server name, username, and password into the Login Form window. The Login Form window is called from the `Connect` method and is shown as a modal dialog box.

If you click the OK button in the Login Form window, the `Connect` method tries to connect you to the remote server. If the connection is successful, it also logs you in. Whether login was successful, it calls the `Log` method. This method appends the specified message to the `messageTextBox` control.

If login is successful, the `Connect` method displays the remote server's home directory content on the `remoteDirList` control.

The Form1 Class's Declaration

The `Form1` class has the following declarations part:

```
Private localCurrentDir As String = Directory.GetCurrentDirectory()
Private remoteCurrentDir As String
Private server, userName, password As String
Private ftp As New ftp()
'the size of the file being downloaded/uploaded
Private fileSize As Integer

Private Structure DirectoryItem
  Public name As String
  Public modifiedDate As String
  Public size As String
End Structure
```

Note that the `DirectoryItem` structure represents either a directory or a file.

The Form1 Class's Methods

The following sections describe the Form1 class's methods.

ChangeLocalDir

The ChangeLocalDir method changes the local directory. This method is called when the user double-clicks an item in the localDirList control. The action taken by this method depends on the item activated. It changes directory if the activated item is either the parent directory icon or a folder. If the item is a file, the ChangeLocalDir method calls the UploadFile method.

This is the ChangeLocalDir method:

```
Private Sub ChangeLocalDir()
  'get activated item (the items that was double-clicked
  Dim item As ListViewItem = localDirList.SelectedItems(0)
  If item.Text.Equals("..") Then
    Dim parentDir As DirectoryInfo = Directory.GetParent(localCurrentDir)
    If Not parentDir Is Nothing Then
      localCurrentDir = parentDir.FullName
      SelectLocalDirectory(localCurrentDir)
    End If
  Else
    Directory.SetCurrentDirectory(localCurrentDir)
    Dim fullPath As String = Path.GetFullPath(item.Text)
    If Helper.IsDirectory(fullPath) Then
      localCurrentDir = fullPath
      SelectLocalDirectory(localCurrentDir)
    Else
      UploadFile()
    End If
  End If
End Sub
```

This method begins by obtaining the selected item from localDirList:

```
'get activated item (the items that was double-clicked
Dim item As ListViewItem = localDirList.SelectedItems(0)
```

It then checks whether the item is a parent directory icon. If it is, it constructs a `DirectoryInfo` object for the parent directory of the current directory using the `GetParent` method of the `System.IO.Directory` class:

```
If item.Text.Equals("..") Then
  Dim parentDir As DirectoryInfo = Directory.GetParent(localCurrentDir)
```

If the `GetParent` method returns a non-null value, it sets `localCurrentDir` to the parent directory's full name and calls the `SelectLocalDirectory` method to repopulate the `localDirList` control:

```
If Not parentDir Is Nothing Then
  localCurrentDir = parentDir.FullName
  SelectLocalDirectory(localCurrentDir)
End If
```

If the activated item is not a parent directory icon, it sets the application current directory to the value of `localCurrentDir` so that it can get the full path to the currently activated item:

```
Else
  Directory.SetCurrentDirectory(localCurrentDir)
  Dim fullPath As String = Path.GetFullPath(item.Text)
```

Now, it has to determine whether the activated item is a file or a directory using the `Helper` class's `IsDirectory` method. If it is a directory, it sets `localCurrentDir` to the full path obtained from the `GetFullPath` method in the previous line and then calls the `SelectLocalDirectory` to repopulate the `localDirList` control:

```
If Helper.IsDirectory(fullPath) Then
  localCurrentDir = fullPath
  SelectLocalDirectory(localCurrentDir)
```

If the activated item is a file, the `ChangeLocalDir` method simply calls the `UploadFile` method:

```
UploadFile()
```

ChangeRemoteDir

The `ChangeRemoteDir` method changes the directory in the connected server:

```
Private Sub ChangeRemoteDir()
  If ftp.Connected Then

    'get activated item (the item that was double-clicked)
    Dim item As ListViewItem = remoteDirList.SelectedItems(0)

    If item.Text.Equals("..") Then
      Dim index As Integer 'get the last index of "/"
      index = remoteCurrentDir.LastIndexOf("/")
      If index = 0 Then
        remoteCurrentDir = "/"
      Else
        remoteCurrentDir = remoteCurrentDir.Substring(0, index)
      End If
      ftp.ChangeDir(remoteCurrentDir)
      If ftp.replyCode.StartsWith("2") Then 'successful
        SelectRemoteDirectory(remoteCurrentDir)
      End If
      Log(ftp.replyMessage)
    ElseIf Helper.IsDirectoryItem(remoteDirList.SelectedItems(0)) Then
      If remoteCurrentDir.Equals("/") Then
        remoteCurrentDir += item.Text
      Else
        remoteCurrentDir += "/" & item.Text
      End If
      ftp.ChangeDir(remoteCurrentDir)
      If ftp.replyCode.StartsWith("2") Then 'successful
        SelectRemoteDirectory(remoteCurrentDir)
      End If
      Log(ftp.replyMessage)
    Else
      DownloadFile()
    End If

  Else
    NotConnected()
  End If
End Sub
```

First, the method only executes the code in its body if a remote server is connected. Checking a connection is through the `FTP` class's `Connected` property:

```
If ftp.Connected Then
   ...
Else
  NotConnected()
End If
```

After making sure that a remote server is connected, it obtains the activated item from the `remoteDirList` control:

```
'get activated item (the item that was double-clicked)
Dim item As ListViewItem = remoteDirList.SelectedItems(0)
```

The action taken by this method depends on the activated item. If the item is a parent directory icon or a folder, it calls the `FTP` class's `ChangeDir` method. If the activated item is a file, it calls the `FTP` class's `Download` method.

Note that because you are dealing with a remote server, you do not have access to the directory system. Instead, you work with paths.

If the activated item is the parent directory icon, the method tries to obtain the parent directory of the remote current directory. The `remoteCurrentDir` variable holds the remote current directory. Obtaining the parent directory is by trimming the characters after the last / (assuming that the remote server uses a Unix directory listing):

```
If item.Text.Equals("..") Then
  Dim index As Integer 'get the last index of "/"
  index = remoteCurrentDir.LastIndexOf("/")
  If index = 0 Then
    ' we are already in the root
    remoteCurrentDir = "/"
  Else
    remoteCurrentDir = remoteCurrentDir.Substring(0, index)
  End If
```

This gives you a new remote current directory. You just need to call the `FTP` class's `ChangeDir` method and pass the new directory name:

```
ftp.ChangeDir(remoteCurrentDir)
```

Now, if the ChangeDir method returns a successful reply message, you call the SelectRemoteDirectory to repopulate the remoteDirList control. You also log the reply message from the server:

```
If ftp.replyCode.StartsWith("2") Then 'successful
  SelectRemoteDirectory(remoteCurrentDir)
End If
Log(ftp.replyMessage)
```

If the activated item is a folder, the user clicks a subdirectory in the current remote directory. You can obtain the new current directory by appending the item's text to the current directory. Note, however, if you are currently in the root, you do not append a / before appending the directory name:

```
ElseIf Helper.IsDirectoryItem(remoteDirList.SelectedItems(0)) Then
  If remoteCurrentDir.Equals("/") Then
    remoteCurrentDir += item.Text
  Else
    remoteCurrentDir += "/" & item.Text
  End If
```

Having a new directory, you call the FTP class's ChangeDir, passing the destination directory:

```
ftp.ChangeDir(remoteCurrentDir)
```

If the ChangeDir method returns a server's successful message, you call the SelectRemoteDirectory to repopulate the remoteDirList control. You also log the server's reply message:

```
If ftp.replyCode.StartsWith("2") Then 'successful
  SelectRemoteDirectory(remoteCurrentDir)
End If
Log(ftp.replyMessage)
```

If the activated item is a file, you call the DownloadFile method:

```
DownloadFile()
```

Connect

The `Connect` method connects to and disconnects from a remote server. If the `connectFileMenuItem`'s `Text` displays *Disconnect*, the application may be connected to a remote server. Because the remote server can disconnect a client if the client is idle for a given period of time, it is possible that the application is not connected to any server even though the client thinks it is still connected. Either way, the `Connect` method handles the situation well. The definition of the `Connect` method is as follows:

```
Private Sub Connect()
  'connect and disconnect
  If connectFileMenuItem.Text.Equals("&Disconnect") Then
    'disconnect
    If MessageBox.Show("Disconnect from remote server?", "Disconnect", _
      MessageBoxButtons.OKCancel, MessageBoxIcon.Question) = _
      DialogResult.OK Then
      If ftp.Connected Then
        ftp.Disconnect()
        Log("Disconnected.")
        connectFileMenuItem.Text = "&Connect"
        'clearing the ListView
        'don't use the remoteDirList.Clear because it removes the columns too,
        'instead use remoteDirList.Items.Clear()
        remoteDirList.Items.Clear()
        'clearing the combo box
        remoteDir.Items.Clear()
        remoteDir.Text = ""
      End If
    End If
  Else
    'connect
    Dim loginForm As New LoginForm()
    Dim loggedIn As Boolean = False
    While Not loggedIn AndAlso loginForm.ShowDialog() = DialogResult.OK
      server = loginForm.Server
      userName = loginForm.UserName
      password = loginForm.Password
      Log("Connecting " & server)
      Try
        ftp.Connect(server)
        If ftp.Connected Then
          Log(server & " connected. Try to login.")
```

```
            If ftp.Login(userName, password) Then
              connectFileMenuItem.Text = "&Disconnect"
              Log("Login successful.")
              loggedIn = True
              ' try to get the remote list
              ftp.ChangeToAsciiMode()
              remoteCurrentDir = ftp.GetCurrentRemoteDir()
              If Not remoteCurrentDir Is Nothing Then
                SelectRemoteDirectory(remoteCurrentDir)
              End If
            Else
              Log("Login failed.")
            End If
          Else
            Log("Connection failed")
          End If
        Catch e As Exception
          Log(e.ToString())
        End Try
      End While
      If Not loggedIn AndAlso _
        Not ftp Is Nothing AndAlso _
        ftp.Connected Then
        ftp.Disconnect()
      End If
    End If
End Sub
```

The `Connect` method first checks the `connectFileMenuItem`'s `Text` property. If it is `&Disconnect`, it tries to disconnect from the connected remote server:

```
If connectFileMenuItem.Text.Equals("&Disconnect") Then
  'disconnect
```

Before disconnecting, it asks for the user's confirmation:

```
If MessageBox.Show("Disconnect from remote server?", "Disconnect", _
  MessageBoxButtons.OKCancel, MessageBoxIcon.Question) = _
  DialogResult.OK Then
```

If the user says OK, it checks if the application is really connected to a remote server. If it is, it calls the `FTP` class's `Disconnect` method, logs a message, changes the `connectFileMenuItem`'s `Text` to `&Connect`, and clears the `remoteDirList` and `remoteDir` controls:

```
If ftp.Connected Then
  ftp.Disconnect()
  Log("Disconnected.")
  connectFileMenuItem.Text = "&Connect"
  'clearing the ListView
  'don't use the remoteDirList.Clear because it removes the columns too,
  'instead use remoteDirList.Items.Clear()
  remoteDirList.Items.Clear()
  'clearing the combo box
  remoteDir.Items.Clear()
  remoteDir.Text = ""
End If
```

If no server is connected, the `Connect` method tries to connect. It starts by defining a `Boolean` called `loggedIn` and showing the Login Form window as a modal dialog box. It then does a `While` loop that loops until one of the following conditions is satisfied:

- The user clicks the Cancel button on the Login Form window.
- The user clicks the OK button on the Login Form window and logs in successfully. This is the code that does that:

```
'connect
Dim loginForm As New LoginForm()
Dim loggedIn As Boolean = False
While Not loggedIn AndAlso loginForm.ShowDialog() = DialogResult.OK
```

When the user clicks the OK button, it takes the values of the Login Form window's `Server`, `UserName`, and `Password` properties and logs a message:

```
server = loginForm.Server
userName = loginForm.UserName
password = loginForm.Password
Log("Connecting " & server)
```

It then tries to connect to the specified server using the FTP class's Connect method:

```
Try
  ftp.Connect(server)
```

If the connection is successful, it logs a message and tries to log in using the FTP class's Login method, passing the username and password:

```
If ftp.Connected Then
  Log(server & " connected. Try to login.")
  If ftp.Login(userName, password) Then
```

The Login method returns True if the user logs in successfully and False otherwise. For a successful login, you change the connectFileMenuItem's Text property to Disconnect, log a successful login message, and change the transfer mode by calling the ChangeToAsciiMode of the FTP class:

```
connectFileMenuItem.Text = "&Disconnect"
Log("Login successful.")
loggedIn = True
' try to get the remote list
ftp.ChangeToAsciiMode()
```

Next, it tries to obtain the remote current directory by calling the GetCurrentRemoteDir method of the FTP class:

```
remoteCurrentDir = ftp.GetCurrentRemoteDir()
```

If this method executes successfully on the remote server, it should return a non-null value, the remote current directory. You then use it as an argument to the SelectRemoteDirectory that displays the content of the remote current directory:

```
If Not remoteCurrentDir Is Nothing Then
  SelectRemoteDirectory(remoteCurrentDir)
End If
```

If the FTP class's Login method you called returns False, you log a "Login failed" message:

```
Log("Login failed.")
```

DeleteLocalFile

The `DeleteLocalFile` method deletes a local file and is called when the user selects an item in the `localDirList` control and clicks the `localDeleteButton` control. The definition of this method is as follows:

```
Private Sub DeleteLocalFile()
  Dim selectedItemCount As Integer = localDirList.SelectedItems.Count
  If selectedItemCount = 0 Then
    MessageBox.Show("Please select a file/directory to delete.", _
      "Warning", MessageBoxButtons.OK, MessageBoxIcon.Warning)
  Else
    If MessageBox.Show("Delete the selected file/directory?", _
      "Delete Confirmation", _
      MessageBoxButtons.OKCancel, MessageBoxIcon.Question) _
      = DialogResult.OK Then
      Dim completePath As String = _
        Path.Combine(localCurrentDir, localDirList.SelectedItems(0).Text)
      Try
        If Helper.IsDirectory(completePath) Then
          Directory.Delete(completePath)
        Else
          File.Delete(completePath)
        End If
        LoadLocalDirList()
      Catch ex As Exception
        MessageBox.Show(ex.ToString(), "Error", MessageBoxButtons.OK, _
          MessageBoxIcon.Error)
      End Try
    End If
  End If

End Sub
```

The `DeleteLocalFile` starts by checking if an item is selected in the `localDirList` control. If not, it displays a message box:

```
Dim selectedItemCount As Integer = localDirList.SelectedItems.Count
If selectedItemCount = 0 Then
  MessageBox.Show("Please select a file/directory to delete.", _
    "Warning", MessageBoxButtons.OK, MessageBoxIcon.Warning)
```

If an item is selected, the `DeleteLocalFile` method asks for the user confirmation that the user intends to delete the item to make sure that the user did not click the `localDeleteButton` control by accident:

```
If MessageBox.Show("Delete the selected file/directory?", _
  "Delete Confirmation", _
  MessageBoxButtons.OKCancel, MessageBoxIcon.Question) _
  = DialogResult.OK Then
```

If deletion is confirmed, it tries to obtain the complete path to the item by combining the local current directory and the item's text:

```
Dim completePath As String = _
  Path.Combine(localCurrentDir, localDirList.SelectedItems(0).Text)
```

Then, it checks whether it is a directory or a file. If it is a directory, it calls the `Delete` method of the `System.IO.Directory` class. If it is a file, the `Delete` method of the `System.IO.File` class is invoked:

```
If Helper.IsDirectory(completePath) Then
  Directory.Delete(completePath)
Else
  File.Delete(completePath)
End If
```

After deletion, the `localDirList` is refreshed by calling the `LoadLocalDirList` method:

```
LoadLocalDirList()
```

DeleteRemoteFile

The `DeleteRemoteFile` method deletes a file on the connected remote server:

```
Private Sub DeleteRemoteFile()
  If ftp.Connected Then
    Dim selectedItemCount As Integer = remoteDirList.SelectedItems.Count
    If selectedItemCount = 0 Then
      MessageBox.Show("Please select a file/directory to delete.", _
        "Warning", MessageBoxButtons.OK, MessageBoxIcon.Warning)
```

```
    Else
      If MessageBox.Show("Delete the selected file/directory?", _
        "Delete Confirmation", _
        MessageBoxButtons.OKCancel, MessageBoxIcon.Question) _
        = DialogResult.OK Then
        Try
          Dim selectedItem As ListViewItem = remoteDirList.SelectedItems(0)
          If Helper.IsDirectoryItem(selectedItem) Then
            If ftp.DeleteDir(selectedItem.Text) Then
              LoadRemoteDirList()
            Else
              Log(ftp.replyMessage)
            End If
          Else
            If ftp.DeleteFile(selectedItem.Text) Then
              LoadRemoteDirList()
            Else
              Log(ftp.replyMessage)
            End If
          End If
        Catch ex As Exception
          MessageBox.Show(ex.ToString, "Error", MessageBoxButtons.OK, _
            MessageBoxIcon.Error)
        End Try
      End If
    End If
  Else
    NotConnected()
  End If
End Sub
```

Note that the method only executes its body if the application is connected to a remote FTP server:

```
If ftp.Connected Then
   ...
Else
  NotConnected()
End If
```

After making sure that a remote server is connected, it checks that an item is selected in the `remoteDirList` control. If no item is selected, a warning displays in a message box:

```
Dim selectedItemCount As Integer = remoteDirList.SelectedItems.Count
If selectedItemCount = 0 Then
  MessageBox.Show("Please select a file/directory to delete.", _
    "Warning", MessageBoxButtons.OK, MessageBoxIcon.Warning)
```

If an item is selected, it asks for the user's confirmation to make sure that the `remoteDeleteButton` control was not clicked by accident:

```
Else
  If MessageBox.Show("Delete the selected file/directory?", _
    "Delete Confirmation", _
    MessageBoxButtons.OKCancel, MessageBoxIcon.Question) _
    = DialogResult.OK Then
```

If deletion is confirmed, the method obtains the selected item and sends it to the `Helper` class's `IsDirectoryItem` to determine if the selected item is a directory or a file:

```
Try
  Dim selectedItem As ListViewItem = remoteDirList.SelectedItems(0)
  If Helper.IsDirectoryItem(selectedItem) Then
```

If the selected item is a directory, it calls the `FTP` class's `DeleteDir` method and, upon successful completion of this method, calls the `LoadRemoteDirList` to repopulate the `remoteDirList` control. If the `DeleteDir` method returns `False` to indicate that the deletion failed, it logs the message:

```
If ftp.DeleteDir(selectedItem.Text) Then
  LoadRemoteDirList()
Else
  Log(ftp.replyMessage)
End If
```

If the selected item is a file, it calls the `FTP` class's `DeleteFile` method and, upon successful completion of this method, calls the `LoadRemoteDirList` to repopulate the `remoteDirList` control. If the `DeleteFile` method failed, it logs the message:

```
If ftp.DeleteFile(selectedItem.Text) Then
  LoadRemoteDirList()
```

```
Else
  Log(ftp.replyMessage)
End If
```

DownloadFile

The `DownloadFile` method downloads a file from the connected remote server:

```
Private Sub DownloadFile()
  If ftp.Connected Then
    Dim selectedItemCount As Integer = remoteDirList.SelectedItems.Count

    If selectedItemCount = 0 Then
      MessageBox.Show("Please select a file to download.", _
        "Warning", MessageBoxButtons.OK, MessageBoxIcon.Warning)
    Else
      Dim item As ListViewItem = remoteDirList.SelectedItems(0)
      If Helper.IsDirectoryItem(item) Then
        MessageBox.Show("You cannot download a directory.", _
          "Error downloading file", MessageBoxButtons.OK, _
          MessageBoxIcon.Error)
      Else
        Try
          fileSize = Convert.ToInt32(item.SubItems(1).Text)
        Catch
        End Try
        ftp.Download(item.Text, localCurrentDir)
      End If
    End If
  Else
    NotConnected()
  End If
End Sub
```

Note that the method only executes its body if the application is connected to a remote FTP server:

```
If ftp.Connected Then
   ...
Else
  NotConnected()
End If
```

After making sure that a remote server is connected, it checks that an item is selected in the `remoteDirList` control. If no item is selected, a warning displays:

```
Dim selectedItemCount As Integer = remoteDirList.SelectedItems.Count
If selectedItemCount = 0 Then
  MessageBox.Show("Please select a file to download.", _
    "Warning", MessageBoxButtons.OK, MessageBoxIcon.Warning)
```

If an item is selected, it gets the selected item from the `remoteDirList` control and sends the item to the `Helper` class's `IsDirectoryItem` method to determine whether the selected item is a directory or a file:

```
Dim item As ListViewItem = remoteDirList.SelectedItems(0)
```

If the item is a directory, the method shows an error message, warning the user that they cannot download a directory:

```
If Helper.IsDirectoryItem(item) Then
  MessageBox.Show("You cannot download a directory.", _
    "Error downloading file", MessageBoxButtons.OK, _
    MessageBoxIcon.Error)
```

If the selected item is a file, it gets the file size from the item and assigns it to the `fileSize` variable. The `FTP` class's `BeginDownload` event handler uses this value to calculate the transfer progress:

```
Try
  fileSize = Convert.ToInt32(item.SubItems(1).Text)
Catch
End Try
```

It then calls the `FTP` class's `Download` method, passing the filename and the local current directory. These two arguments determine where to save the downloaded file:

```
ftp.Download(item.Text, localCurrentDir)
```

GetDirectoryItem

This FTP client application only works properly if the remote server returns a directory listing in Unix style. If this is the case, the list contains lines of data that include the directory/filename and each directory/file's meta information. The

`GetDirectoryItem` method processes the line that contains a directory/file information and returns it as a `DirectoryItem` object. The resulting `DirectoryItem` object can contain information about a directory or a file.

The raw data passed to this method has the following format:

```
-rwxrwxrwx   1 owner    group           11801 Jul 23 10:52 NETFTP.vb
```

or this format:

```
drwxrwxrwx   1 owner    group               0 Jul 26 20:11 New Folder
```

The method definition is as follows:

```
Private Function GetDirectoryItem(ByVal s As String) As DirectoryItem
  's is in the following format
  '-rwxrwxrwx   1 owner    group           11801 Jul 23 10:52 NETFTP.vb
  '
  'or
  '
  'drwxrwxrwx   1 owner    group               0 Jul 26 20:11 New Folder

  Dim dirItem As New DirectoryItem()
  If Not s Is Nothing Then
    Dim index As Integer
    index = s.IndexOf(" "c)
    If index <> -1 Then
      s = s.Substring(index).TrimStart() 'removing "drwxrwxrwx" part
      'now s is in the following format
      '1 owner    group           11801 Jul 23 10:52 NETFTP.vb
      '
      'or
      '
      '1 owner    group               0 Jul 26 20:11 New Folder
      index = s.IndexOf(" "c)
      If index <> -1 Then
        s = s.Substring(index).TrimStart() 'removing the '1' part
        'now s is in the following format
        'owner    group           11801 Jul 23 10:52 NETFTP.vb
        '
        'or
        '
        'owner    group               0 Jul 26 20:11 New Folder
        index = s.IndexOf(" "c)
```

```
If index <> -1 Then
  s = s.Substring(index).TrimStart() 'removing the 'owner' part
  'now s is in the following format
  'group           11801 Jul 23 10:52 NETFTP.vb
  '
  'or
  '
  'group               0 Jul 26 20:11 New Folder
  index = s.IndexOf(" "c)
  If index <> -1 Then
    s = s.Substring(index).TrimStart() 'removing the 'group' part
    'now s is in the following format
    '11801 Jul 23 10:52 NETFTP.vb
    '
    'or
    '
    '0 Jul 26 20:11 New Folder
    'now get the size.
    index = s.IndexOf(" "c)
    If index > 0 Then
      dirItem.size = s.Substring(0, index)
      s = s.Substring(index).TrimStart() 'removing the size
      'now s is in the following format
      'Jul 23 10:52 NETFTP.vb
      '
      'or
      '
      'Jul 26 20:11 New Folder
      'now, get the 3 elements of the date part
      Dim date1, date2, date3 As String
      index = s.IndexOf(" "c)
      If index <> -1 Then
        date1 = s.Substring(0, index)
        s = s.Substring(index).TrimStart()
        index = s.IndexOf(" "c)
        If index <> -1 Then
          date2 = s.Substring(0, index)
          s = s.Substring(index).TrimStart()
          index = s.IndexOf(" "c)
          If index <> -1 Then
            date3 = s.Substring(0, index)
            dirItem.modifiedDate = date1 & " " & date2 & " " & date3
            ' get the name
```

```
                    dirItem.name = s.Substring(index).Trim()
                  End If
                End If
              End If
            End If
          End If
        End If
      End If
    End If
  End If
  Return dirItem
End Function
```

The method starts by constructing a `DirectoryItem` object. This object will be populated and returned to the function's caller:

```
Dim dirItem As New DirectoryItem()
```

First the method checks that `s` (the argument passed to this method) is not null:

```
If Not s Is Nothing Then
```

If `s` is not null, then the method finds the first space in `s`, modifies `s` so that `s` does not include the string before the space, and left-trims `s` until the next non-space character:

```
Dim index As Integer
index = s.IndexOf(" "c)
      If index <> -1 Then
  s = s.Substring(index).TrimStart() 'removing "drwxrwxrwx" part
```

`s` now has the following format:

```
1 owner    group           11801 Jul 23 10:52 NETFTP.vb
```

or this format:

```
1 owner    group               0 Jul 26 20:11 New Folder
```

Then, you do the same thing as you did just now:

```
index = s.IndexOf(" "c)
If index <> -1 Then
  s = s.Substring(index).TrimStart() 'removing the '1' part
```

to get s in the following format:

```
owner    group           11801 Jul 23 10:52 NETFTP.vb
```

or this format:

```
owner    group               0 Jul 26 20:11 New Folder
```

And again:

```
index = s.IndexOf(" "c)
If index <> -1 Then
  s = s.Substring(index).TrimStart() 'removing the 'owner' part
```

to get s in the following format:

```
group           11801 Jul 23 10:52 NETFTP.vb
```

or this format:

```
group               0 Jul 26 20:11 New Folder
```

And yet another one:

```
index = s.IndexOf(" "c)
If index <> -1 Then
  s = s.Substring(index).TrimStart() 'removing the 'group' part
```

Now, s has the following format:

```
11801 Jul 23 10:52 NETFTP.vb
```

or this format:

```
0 Jul 26 20:11 New Folder
```

Now, you can get the size and do the same operation:

```
index = s.IndexOf(" "c)
If index > 0 Then
  dirItem.size = s.Substring(0, index)
  s = s.Substring(index).TrimStart() 'removing the size
```

Afterward, `s` has the following format:

```
Jul 23 10:52 NETFTP.vb
```

or this one:

```
Jul 26 20:11 New Folder
```

Now, get the three elements of the date part:

```
Dim date1, date2, date3 As String
index = s.IndexOf(" "c)
If index <> -1 Then
  date1 = s.Substring(0, index)
  s = s.Substring(index).TrimStart()
  index = s.IndexOf(" "c)
  If index <> -1 Then
    date2 = s.Substring(0, index)
     s = s.Substring(index).TrimStart()
    index = s.IndexOf(" "c)
    If index <> -1 Then
      date3 = s.Substring(0, index)
      dirItem.modifiedDate = date1 & " " & date2 & " " & date3
      ' get the name
      dirItem.name = s.Substring(index).Trim()
```

Finally, return the `DirectoryItem` object:

```
  Return dirItem
```

InitializeProgressBar

The `InitializeProgressBar` method initializes the progress bar prior to file transfer:

```
Private Sub InitializeProgressBar()
  progressBar.Value = 0
  progressBar.Maximum = fileSize
End Sub
```

The `InitializeProgressBar` method sets the `Value` property to 0 and the `Maximum` property to `fileSize`. `fileSize` contains the number of bytes to transfer:

```
progressBar.Value = 0
progressBar.Maximum = fileSize
```

LoadLocalDirList

The `LoadLocalDirList` method populates the `localDirList` control with the content of the current directory. If the current directory is not the root, an icon representing a parent directory is also added. The method definition is as follows:

```
Private Sub LoadLocalDirList()
  localDirList.Items.Clear()
  Dim item As ListViewItem

  ' if current directory is not root, add pointer to parent dir
  If Not Directory.GetParent(localCurrentDir) Is Nothing Then
    item = New ListViewItem("..", 1)
    item.ImageIndex = 0
    localDirList.Items.Add(item)
  End If

  ' list of directories
  Dim directories As String() = Directory.GetDirectories(localCurrentDir)
  Dim length As Integer = directories.Length
  Dim dirName As String
  For Each dirName In directories
    item = New ListViewItem(Path.GetFileName(dirName), 1)
    item.SubItems.Add("")
    item.SubItems.Add(Directory.GetLastAccessTime(dirName).ToString())
    item.ImageIndex = 1
    localDirList.Items.Add(item)
```

```
  Next
  'list of files
  Dim files As String() = Directory.GetFiles(localCurrentDir)
  length = files.Length
  Dim fileName As String
  For Each fileName In files
    item = New ListViewItem(Path.GetFileName(fileName), 1)
    Dim fi As New FileInfo(fileName)
    item.SubItems.Add(Convert.ToString(fi.Length))
    item.SubItems.Add(File.GetLastWriteTime(fileName).ToString())
    item.ImageIndex = 2
    localDirList.Items.Add(item)
  Next

End Sub
```

The method starts by clearing the `localDirList` control and defining a `ListViewItem` called `item`:

```
localDirList.Items.Clear()
Dim item As ListViewItem
```

If the current directory is not root, it adds a parent directory icon:

```
If Not Directory.GetParent(localCurrentDir) Is Nothing Then
  item = New ListViewItem("..", 1)
  item.ImageIndex = 0
  localDirList.Items.Add(item)
End If
```

Next, it adds all subdirectories in the current directory. You obtain the list of directories from the `GetDirectories` method of the `System.IO.Directory` class:

```
Dim directories As String() = Directory.GetDirectories(localCurrentDir)
Dim length As Integer = directories.Length
Dim dirName As String
For Each dirName In directories
  item = New ListViewItem(Path.GetFileName(dirName), 1)
  item.SubItems.Add("")
  item.SubItems.Add(Directory.GetLastAccessTime(dirName).ToString())
  item.ImageIndex = 1
  localDirList.Items.Add(item)
Next
```

Finally, it adds all files in the current directory. You obtain the list of files from the `GetFiles` method of the `Directory` class:

```
Dim files As String() = Directory.GetFiles(localCurrentDir)
length = files.Length
Dim fileName As String
For Each fileName In files
  item = New ListViewItem(Path.GetFileName(fileName), 1)
  Dim fi As New FileInfo(fileName)
  item.SubItems.Add(Convert.ToString(fi.Length))
  item.SubItems.Add(File.GetLastWriteTime(fileName).ToString())
  item.ImageIndex = 2
  localDirList.Items.Add(item)
Next
```

LoadRemoteDirList

The `LoadRemoteDirList` method populates the `remoteDirList` control with the content of the remote current directory. If the remote current directory is not the root, an icon representing a parent directory is also added. The definition of the `LoadRemoteDirList` is as follows:

```
Private Sub LoadRemoteDirList()
  If ftp.Connected Then
    remoteDirList.Items.Clear()
    Dim item As ListViewItem

    If Not remoteCurrentDir.Equals("/") Then
      item = New ListViewItem("..", 1)
      item.ImageIndex = 0
      remoteDirList.Items.Add(item)
    End If

    Try
      ftp.ChangeDir(remoteCurrentDir)
      ftp.GetDirList()
      Dim lines As String() = _
        ftp.DirectoryList.Split(Convert.ToChar(ControlChars.Cr))
      Dim line As String
      Dim fileList As New ArrayList()
      Dim dirList As New ArrayList()
      For Each line In lines
        If line.Trim().StartsWith("-") Then ' a file
```

```
            fileList.Add(line)
          ElseIf line.Trim().StartsWith("d") Then ' a directory
            dirList.Add(line)
          End If
        Next

        ' now load subdirectories to DirListView
        Dim enumerator As IEnumerator = dirList.GetEnumerator
        While enumerator.MoveNext
          Dim dirItem As DirectoryItem = _
            GetDirectoryItem(CType(enumerator.Current, String))
          If Not dirItem.name Is Nothing Then
            item = New ListViewItem(dirItem.name, 1)
            item.SubItems.Add("")
            item.SubItems.Add(dirItem.modifiedDate)
            remoteDirList.Items.Add(item)
          End If
        End While

        enumerator = fileList.GetEnumerator
        While enumerator.MoveNext
          Dim dirItem As DirectoryItem = _
            GetDirectoryItem(CType(enumerator.Current, String))
          If Not dirItem.name Is Nothing Then
            item = New ListViewItem(dirItem.name, 2)
            item.SubItems.Add(diritem.size)
            item.SubItems.Add(dirItem.modifiedDate)
            remoteDirList.Items.Add(item)
          End If
        End While
      Catch e As Exception
        Debug.WriteLine(e.ToString())
      End Try
    Else
      NotConnected()
    End If
  End Sub
```

The method starts by checking if the application is connected to a remote server. It only executes the rest of the code in its body if the application is connected:

```
If ftp.Connected Then
  ...
Else
  NotConnected()
End If
```

If the application is connected, the `remoteDirList` control is cleared and a `ListViewItem` variable called item is defined:

```
remoteDirList.Items.Clear()
Dim item As ListViewItem
```

If the remote current directory is not root, the method adds the icon representing the parent directory:

```
If Not remoteCurrentDir.Equals("/") Then
  item = New ListViewItem("..", 1)
  item.ImageIndex = 0
  remoteDirList.Items.Add(item)
End If
```

Next, it changes directory to the remote current directory and calls the `FTP` class's `GetDirList` to obtain the directory listing:

```
Try
  ftp.ChangeDir(remoteCurrentDir)
  ftp.GetDirList()
```

The directory listing returns in a long string containing file and directory information in the remote current directory. The string then splits into lines:

```
Dim lines As String() = _
  ftp.DirectoryList.Split(Convert.ToChar(ControlChars.Cr))
```

The subdirectories and files returned are not grouped by type, but you want to display subdirectories in one group and files in another. Therefore, you construct an `ArrayList` called `fileList` that will hold the list of files and an `ArrayList` named `dirList` to hold the list of directories:

```
Dim line As String
Dim fileList As New ArrayList()
Dim dirList As New ArrayList()
```

Then, each line that starts with a hyphen in a file is added to `fileList` and lines starting with *d* are added to `dirList`:

```
For Each line In lines
  If line.Trim().StartsWith("-") Then ' a file
    fileList.Add(line)
  ElseIf line.Trim().StartsWith("d") Then ' a directory
    dirList.Add(line)
  End If
Next
```

Now, you can add all subdirectories to the `remoteDirList` control. Note that you use the `GetDirectoryItem` to convert the raw text to a `DirectoryItem` object:

```
Dim enumerator As IEnumerator = dirList.GetEnumerator
While enumerator.MoveNext
  Dim dirItem As DirectoryItem = _
    GetDirectoryItem(CType(enumerator.Current, String))
  If Not dirItem.name Is Nothing Then
    item = New ListViewItem(dirItem.name, 1)
    item.SubItems.Add("")
    item.SubItems.Add(dirItem.modifiedDate)
    remoteDirList.Items.Add(item)
  End If
End While
```

Next, you add all files to the `remoteDirList` control, again using the `GetDirectoryItem` method to get `DirectoryItem` objects:

```
enumerator = fileList.GetEnumerator
While enumerator.MoveNext
  Dim dirItem As DirectoryItem = _
    GetDirectoryItem(CType(enumerator.Current, String))
  If Not dirItem.name Is Nothing Then
    item = New ListViewItem(dirItem.name, 2)
    item.SubItems.Add(diritem.size)
    item.SubItems.Add(dirItem.modifiedDate)
    remoteDirList.Items.Add(item)
  End If
End While
```

Log

The `Log` method appends the text passed to it to the `messageTextBox` control's `Text` property and forces the `messageTextBox` control to scroll to the end of the text:

```
Private Sub Log(ByVal message As String)
  messageTextBox.Text += message & ControlChars.CrLf
  'forces the TextBox to scroll
  messageTextBox.SelectionStart = messageTextBox.Text.Length
  messageTextBox.ScrollToCaret()
End Sub
```

MakeLocalDir

The `MakeLocalDir` method creates a new directory under the local current directory:

```
Private Sub MakeLocalDir()
  Dim dirName As String = InputBox( _
    "Enter the name of the directory to create in the local computer", _
    "Make New Directory").Trim()
  If Not dirName.Equals("") Then
    Dim fullPath As String = Path.Combine(localCurrentDir, dirName)
    If Directory.Exists(fullPath) Then
      MessageBox.Show("Directory already exists.", _
        "Error creating directory", MessageBoxButtons.OK, _
        MessageBoxIcon.Error)
    Else
      If File.Exists(fullPath) Then
        MessageBox.Show("Directory name is the same as the name of a file.", _
          "Error creating directory", MessageBoxButtons.OK, _
          MessageBoxIcon.Error)
      Else
        Try
          Directory.CreateDirectory(fullPath)
          LoadLocalDirList()
        Catch e As Exception
          MessageBox.Show(e.ToString, _
            "Error creating directory", MessageBoxButtons.OK, _
            MessageBoxIcon.Error)
        End Try
      End If
    End If
  End If
End Sub
```

It begins by prompting the user to enter a name for the new directory:

```
Dim dirName As String = InputBox( _
  "Enter the name of the directory to create in the local computer", _
  "Make New Directory").Trim()
```

If the user enters a valid name, it gets the full path by combining the local current directory and the entered name:

```
If Not dirName.Equals("") Then
  Dim fullPath As String = Path.Combine(localCurrentDir, dirName)
```

Next, it checks if the directory already exists:

```
If Directory.Exists(fullPath) Then
  MessageBox.Show("Directory already exists.", _
    "Error creating directory", MessageBoxButtons.OK, _
    MessageBoxIcon.Error)
```

It also checks if the directory name resembles a file in the same directory:

```
If File.Exists(fullPath) Then
  MessageBox.Show("Directory name is the same as the name of a file.", _
    "Error creating directory", MessageBoxButtons.OK, _
    MessageBoxIcon.Error)
```

If the name is unique in the directory, it uses the `CreateDirectory` method of the `System.IO.Directory` class to create a new directory:

```
Directory.CreateDirectory(fullPath)
```

Upon a successful create operation, it refreshes the content of the `localDirList` control by calling the `LoadLocalDirList` method:

```
LoadLocalDirList()
```

MakeRemoteDir

The `MakeRemoteDir` method creates a new directory in the remote current directory. If the application is connected to a remote server, it prompts the user for a directory name. The `MakeRemoteDir` method definition is as follows:

```
Private Sub MakeRemoteDir()
  If ftp.Connected Then
    Dim dirName As String = InputBox( _
      "Enter the name of the directory to create in the remote server", _
      "Make New Directory").Trim()
    If Not dirName.Equals("") Then
      ftp.MakeDir(dirName)
      Log(ftp.replyMessage)
      If ftp.replyCode.StartsWith("2") Then
        LoadRemoteDirList()
        'Dim item As New ListViewItem(dirName, 1)
         'If remoteCurrentDir.Equals("/") Then
        '  remoteDirList.Items.Insert(1, item)
        'Else
        '  remoteDirList.Items.Insert(0, item)
        'End If
      End If
    End If
  Else
    NotConnected()
  End If
End Sub
```

It starts by prompting the user to enter a directory name into an `InputBox`:

```
  If ftp.Connected Then
    Dim dirName As String = InputBox( _
      "Enter the name of the directory to create in the remote server", _
      "Make New Directory").Trim()
```

If the name is not blank, it calls the `FTP` class's `MakeDir` method and logs the message:

```
    If Not dirName.Equals("") Then
      ftp.MakeDir(dirName)
      Log(ftp.replyMessage)
```

If the `MakeDir` method is successful (indicated by a reply code starting with 2), it calls the `LoadRemoteDirList` method:

```
If ftp.replyCode.StartsWith("2") Then
  LoadRemoteDirList()
End If
```

NotConnected

The `NotConnected` method is called every time another method finds out that the application is not connected to a remote server. It logs a message and then changes the `connectFileMenuItem`'s `Text` property to `&Connect`. The method definition is as follows:

```
Private Sub NotConnected()
  Log("Not connected")
  connectFileMenuItem.Text = "&Connect"
  'clearing the ListView
  'don't use the remoteDirList.Clear because it removes the columns too,
  'instead use remoteDirList.Items.Clear()
  remoteDirList.Items.Clear()
  'clearing the combo box
  remoteDir.Items.Clear()
  remoteDir.Text = ""
End Sub
```

The first thing it does is to log the "Not connected" message and change the `connectFileMenuItem`'s `Text` property:

```
Log("Not connected")
connectFileMenuItem.Text = "&Connect"
```

It then clears the `remoteDirList` and `remoteDir` controls:

```
remoteDirList.Items.Clear()
'clearing the combo box
remoteDir.Items.Clear()
remoteDir.Text = ""
```

RenameLocalFile

The `RenameLocalFile` method renames a file in the local computer. This method is invoked when the user clicks the `localRenameButton` control. The method definition is as follows:

```
Private Sub RenameLocalFile()
  Dim selectedItemCount As Integer = localDirList.SelectedItems.Count
  If selectedItemCount = 0 Then
    MessageBox.Show("Please select a file/directory to rename.", _
      "Warning", MessageBoxButtons.OK, MessageBoxIcon.Warning)
  Else
    Dim newName As String = InputBox("Enter the new name", "Rename").Trim()
    If Not newName.Equals("") Then
      Dim item As ListViewItem = localDirList.SelectedItems(0)
      If newName.Equals(item.Text) Then
        MessageBox.Show("Please enter a different name from the " & _
          "file/directory you are trying to rename.", _
          "Error renaming file/directory", MessageBoxButtons.OK, _
          MessageBoxIcon.Error)
      Else
        Dim fullPath As String = Path.Combine(localCurrentDir, item.Text)
        If Helper.IsDirectory(fullPath) Then
          Directory.Move(fullPath, Path.Combine(localCurrentDir, newName))
        Else
          Dim fi As New FileInfo(fullPath)
          fi.MoveTo(Path.Combine(localCurrentDir, newName))
        End If
        LoadLocalDirList()
      End If
    End If
  End If
End Sub
```

It first checks if an item is selected in the `localDirList` control. If not, it shows a warning:

```
Dim selectedItemCount As Integer = localDirList.SelectedItems.Count
If selectedItemCount = 0 Then
  MessageBox.Show("Please select a file/directory to rename.", _
    "Warning", MessageBoxButtons.OK, MessageBoxIcon.Warning)
```

If an item is selected in the `localDirList` control, it prompts for a new name:

```
Dim newName As String = InputBox("Enter the new name", "Rename").Trim()
```

If the new name does not consist only of spaces, it gets the selected item from the `localDirList` control and checks the old name. If the old name is the same as the new name, it displays an error message:

```
If Not newName.Equals("") Then
  Dim item As ListViewItem = localDirList.SelectedItems(0)
  If newName.Equals(item.Text) Then
    MessageBox.Show("Please enter a different name from the " & _
      "file/directory you are trying to rename.", _
      "Error renaming file/directory", MessageBoxButtons.OK, _
      MessageBoxIcon.Error)
```

If the new name does not conflict with the old one, it composes the full path using the static `Combine` method of the `System.IO.Path` class:

```
Dim fullPath As String = Path.Combine(localCurrentDir, item.Text)
```

Then it checks whether the selected item is a directory or a file. If it is a file, the method calls the `Move` method of the `System.IO.Directory` class to change the name:

```
If Helper.IsDirectory(fullPath) Then
  Directory.Move(fullPath, Path.Combine(localCurrentDir, newName))
```

If the selected item is a file, the method constructs a `FileInfo` object and calls its `MoveTo` method to change the filename:

```
Dim fi As New FileInfo(fullPath)
fi.MoveTo(Path.Combine(localCurrentDir, newName))
```

Finally, it invokes the `LoadLocalDirList` to refresh the content of the `localDirList` control:

```
LoadLocalDirList()
```

RenameRemoteFile

The `RenameRemoteFile` method renames a file on the connected remote server. It only runs the code in its body if the application is connected to a remote server. The method definition is as follows:

```
Private Sub RenameRemoteFile()
  If ftp.Connected Then
    Dim selectedItemCount As Integer = remoteDirList.SelectedItems.Count
    If selectedItemCount = 0 Then
      MessageBox.Show("Please select a file/directory to rename.", _
        "Warning", MessageBoxButtons.OK, MessageBoxIcon.Warning)
    Else
      Dim dirName As String = InputBox( _
        "Enter the new name", "Rename").Trim()
      If Not dirName.Equals("") Then
        Dim item As ListViewItem = remoteDirList.SelectedItems(0)
        If dirName.Equals(item.Text) Then
          MessageBox.Show("Please enter a different name from the " & _
            "file/directory you are trying to rename.", _
            "Error renaming file/directory", MessageBoxButtons.OK, _
            MessageBoxIcon.Error)
        Else
          ftp.Rename(item.Text, dirName)
          If ftp.replyCode.StartsWith("2") Then
            item.Text = dirName
          End If
          Log(ftp.replyCode & " " & ftp.replyMessage)
        End If
      End If
    End If
  Else
    NotConnected()
  End If
End Sub
```

It starts by checking if there is a selected item in the `remoteDirList` control. If there is not, an error message displays:

```
If ftp.Connected Then
  Dim selectedItemCount As Integer = remoteDirList.SelectedItems.Count
  If selectedItemCount = 0 Then
    MessageBox.Show("Please select a file/directory to rename.", _
      "Warning", MessageBoxButtons.OK, MessageBoxIcon.Warning)
```

If there is a selected item, the method prompts the user for a new name:

```
Dim dirName As String = InputBox( _
  "Enter the new name", "Rename").Trim()
```

If the new name does not consists of spaces only, it gets the selected item and gets the old name in the item's `Text` property. It then compares the new name with the old name:

```
If Not dirName.Equals("") Then
  Dim item As ListViewItem = remoteDirList.SelectedItems(0)
```

If the new name equals the old name, it displays an error message:

```
If dirName.Equals(item.Text) Then
  MessageBox.Show("Please enter a different name from the " & _
    "file/directory you are trying to rename.", _
    "Error renaming file/directory", MessageBoxButtons.OK, _
    MessageBoxIcon.Error)
```

Otherwise, it calls the `FTP` class's `Rename` method, passing both the old name and the new name:

```
ftp.Rename(item.Text, dirName)
```

If the `Rename` method successfully executes (indicated by a server reply code starting with 2), the method updates the item's `Text` property in the `remoteDirList` control:

```
If ftp.replyCode.StartsWith("2") Then
  item.Text = dirName
End If
```

Finally, it logs the reply message from the server:

```
Log(ftp.replyCode & " " & ftp.replyMessage)
```

SelectLocalDirectory

The `SelectLocalDirectory` method inserts the local current directory into the `localDir` combo box:

```
Private Sub SelectLocalDirectory(ByVal path As String)
  ' add current dir to the list
  localDir.Items.Remove(path)
  localDir.Items.Insert(0, path)
  'this will trigger the localDir ComboBox's SelectedIndexChanged event
  localDir.SelectedIndex = 0
End Sub
```

Prior to insertion, it removes the same item to avoid duplication:

```
' add current dir to the list
localDir.Items.Remove(path)
localDir.Items.Insert(0, path)
```

The method then sets the `SelectedIndex` property of the `localDir` combo box to 0 to make the new item selected. This also triggers the combo box's `SelectedIndexChanged` event:

```
    localDir.SelectedIndex = 0
```

SelectRemoteDirectory

If the application is connected to a remote server, this method inserts a remote directory name at the first position in the `remoteDir` combo box. The method definition is as follows:

```
Private Sub SelectRemoteDirectory(ByVal path As String)
  If ftp.Connected Then
    ' add current dir to thel ist
    remoteDir.Items.Remove(path)
    remoteDir.Items.Insert(0, path)
    'this will trigger the remoteDir ComboBox's SelectedIndexChanged event
    remoteDir.SelectedIndex = 0
  Else
    NotConnected()
  End If
End Sub
```

The first thing the method does is to remove the same item to avoid duplication:

```
If ftp.Connected Then
  ' add current dir to thel ist
  remoteDir.Items.Remove(path)
  remoteDir.Items.Insert(0, path)
```

It then sets the `SelectedIndex` property to 0. This triggers the `SelectedIndexChanged` event of the `remoteDir` combo box:

```
remoteDir.SelectedIndex = 0
```

UpdateLocalDir

The `UpdateLocalDir` method is invoked when the `localDir` control's `SelectedIndexChanged` event is triggered. This event is raised when the user manually selects a directory from the `localDir` control or the index is changed programmatically. Then, this method refreshes the content of the `localDirList` control. The method definition is as follows:

```
Private Sub UpdateLocalDir()
  Dim selectedIndex As Integer = localDir.SelectedIndex
  If localDir.SelectedIndex <> -1 Then
    localCurrentDir = CType(localDir.Items(selectedIndex), String)
    LoadLocalDirList()
  End If
End Sub
```

UpdateProgressBar

The `UpdateProgressBar` method is called repeatedly by the event handler that handles the `TransferProgressChanged` event. This method updates the value of the progress bar. The method definition is as follows:

```
Private Sub UpdateProgressBar(ByVal count As Integer)
   progressBar.Value = count
 End Sub
```

UpdateRemoteDir

The `UpdateRemoteDir` method is invoked when the `remoteDir` control's `SelectedIndexChanged` event triggers. This event is raised when the user manually selects a directory from the `remoteDir` control or the index is changed programmatically. This is the `UpdateRemoteDir` method:

```
Private Sub UpdateRemoteDir()
  If ftp.Connected Then
    Dim selectedIndex As Integer = remoteDir.SelectedIndex
    If remoteDir.SelectedIndex <> -1 Then
      remoteCurrentDir = CType(remoteDir.Items(selectedIndex), String)
      LoadRemoteDirList()
    End If
  Else
    NotConnected()
  End If
End Sub
```

When the application is connected to a remote server, this method refreshes the content of the `remoteDirList` control.

UploadFile

The `UploadFile` method uploads a file to the connected remote server if the application is connected to a remote server. This is the method definition:

```
Private Sub UploadFile()
  If ftp.Connected Then
    Dim selectedItemCount As Integer = localDirList.SelectedItems.Count
    If selectedItemCount = 0 Then
      MessageBox.Show("Please select a file to upload.", _
        "Warning", MessageBoxButtons.OK, MessageBoxIcon.Warning)
    Else
      Dim item As ListViewItem = localDirList.SelectedItems(0)
      If Helper.IsDirectoryItem(item) Then
        MessageBox.Show("You cannot upload a directory.", _
          "Error uploading file", MessageBoxButtons.OK, _
          MessageBoxIcon.Error)
      Else
        Try
          fileSize = Convert.ToInt32(item.SubItems(1).Text)
        Catch
```

```
        End Try
        ftp.Upload(item.Text, localCurrentDir)
      End If
    End If
  Else
    NotConnected()
  End If
End Sub
```

It first checks if the application is connected to a remote server:

```
If ftp.Connected Then
```

If the application is connected, the method checks if there is a selected item in the `localDirList` control. If there is no item selected, it displays an error message:

```
Dim selectedItemCount As Integer = localDirList.SelectedItems.Count
If selectedItemCount = 0 Then
  MessageBox.Show("Please select a file to upload.", _
    "Warning", MessageBoxButtons.OK, MessageBoxIcon.Warning)
```

If there is an item selected, it gets the selected item from the `localDirList` control:

```
Dim item As ListViewItem = localDirList.SelectedItems(0)
```

It then checks if the item is a directory item. If it is, the method displays an error message:

```
If Helper.IsDirectoryItem(item) Then
  MessageBox.Show("You cannot upload a directory.", _
    "Error uploading file", MessageBoxButtons.OK, _
    MessageBoxIcon.Error)
```

If the selected item is a file item, it assigns the file size to `fileSize` and calls the `FTP` class's `Upload` method:

```
Try
  fileSize = Convert.ToInt32(item.SubItems(1).Text)
Catch
End Try
ftp.Upload(item.Text, localCurrentDir)
```

Compiling and Running the Application

You can find the source files for the application in the chapter's project directory. To compile the application, run the `Build.bat` file. The content of the `Build.bat` file is as follows:

```
vbc /t:library /r:System.Windows.Forms.dll Helper.vb
vbc /t:library /r:System.dll,System.Windows.Forms.dll,System.Drawing.dll
LoginForm.vb
vbc /t:library /r:System.dll FTP.vb
vbc /t:winexe /r:System.dll,System.Windows.Forms.dll,
System.Drawing.dll,Helper.dll,LoginForm.dll,
FTP.dll Form1.vb
```

Summary

In this chapter you learned how to use sockets to connect to a remote server. You also saw the code to resolve a DNS name into an IP address. You also learned about the RFC959 specification and saw how it is implemented in an FTP client application.