Creating an FTP Server in C# - with IPv6 Support

Rick Bassham, 7 Oct 2013

An introduction into creating a working FTP server in C# using the RFC specification.

Download source code - 117.6 KB

Introduction

First, I want to describe what this article is not. It is not an example of a full featured, scalable, and secure FTP server. It is an introduction into creating an application from a specification, an introduction to socket communication, asynchronous methods, stream encodings, and basic encryption. Please do not use this FTP server in any production environment. It is not secure, and I have done no scalability testing on it. With that out of the way, let's get started.

What is FTP?

According to the specification in IETF RFC 959, the File Transfer Protocol has the following objectives:

1. to promote sharing of files (computer programs and/or data)
2. to encourage indirect or implicit (via programs) use of remote computers
3. to shield a user from variations in file storage systems among

hosts, and
4. to transfer data reliably and efficiently.

FTP is a way to transfer files from one computer to another. Typically, a client connects to a server on port 21, sends some login information, and gets access to the server's local filesystem.

Basic steps

We will start by creating a server that can listen for connections from a client. Once we can accept connections, we will learn to pass off those connections to another thread to handle the processing of commands.

How to listen for connections?

The first step in building our FTP server is getting our server to listen for connections from a client. Let's start with the basic `FTPServer` class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Text;
using System.Net.Sockets;
using System.IO;
using System.Threading;

namespace SharpFtpServer
{
    public class FtpServer
    {
        private TcpListener _listener;

        public FtpServer()
        {
        }

        public void Start()
        {
            _listener = new TcpListener(IPAddress.Any,
21);
            _listener.Start();

_listener.BeginAcceptTcpClient(HandleAcceptTcpClient,
_listener);
        }

        public void Stop()
        {
            if (_listener != null)
            {
                _listener.Stop();
```

```
            }
        }

        private void HandleAcceptTcpClient(IAsyncResult
result)
        {
            TcpClient client =
_listener.EndAcceptTcpClient(result);

_listener.BeginAcceptTcpClient(HandleAcceptTcpClient,
_listener);

            // DO SOMETHING.
        }
    }
}
```

Let's break down what is happening here. According to the MSDN documentation, the `TcpListener` "Listens for connections from TCP network clients." We create it and tell it to listen on port 21 for any `IPAddress` on the server. If you have multiple network adapters in your machine, you may want to limit which one your FTP server listens on. If so, just replace `IPAddress.Any` with a reference to the IP address you want to listen on. After creating it, we call `Start` and then `BeginAcceptTcpClient`. `Start` is obvious, it just starts the `TcpListener` to listen for connections. Now that the `TcpListener` is listening for connections, we have to tell it to do something when a client connects. That is exactly what `BeginAcceptTcpClient` does. We are passing in a reference to another method (`HandleAcceptTcpClient`), which is going to do the work for us. `BeginAcceptTcpClient` does not block execution, instead it returns immediately. When someone does connect, the .NET Framework will call the `HandleAcceptTcpClient` method. Once there, we call `_listener.EndAcceptTcpClient`, passing in the `IAsyncResult` we received. This will return a reference to the `TcpClient` we can use to communicate with the client. Finally, we have to tell the `TcpListener` to keep listening for more connections.

We are connected, now what?

Let us start by getting a reference to the `NetworkStream` that exists between the client and the server. In FTP, this initial connection is called the "Control" connection, as it is used to send commands to the server and for the server to send responses back to the client. Any transfer of files is handled later, in the "Data" connection. The Control connection uses a simple, text based way of sending commands and receiving responses based on TELNET, using the ASCII character set. Since we know this, we can create an easy to use `StreamWriter` and `StreamReader` to make communicating

back and forth easy. Up until this point, we haven't really created anything that is an FTP server, more just something that listens on port 21, accepts connections, and can read/write ASCII back and forth. Let's see the reading/writing in action:

```
private void HandleAcceptTcpClient(IAsyncResult result)
{
    TcpClient client =
_listener.EndAcceptTcpClient(result);
    _listener.BeginAcceptTcpClient(HandleAcceptTcpClient,
_listener);

    NetworkStream stream = client.GetStream();

    using (StreamWriter writer = new StreamWriter(stream,
Encoding.ASCII))
    using (StreamReader reader = new StreamReader(stream,
Encoding.ASCII))
    {
        writer.WriteLine("YOU CONNECTED TO ME");
        writer.Flush();
        writer.WriteLine("I will repeat after you. Send a
blank line to quit.");
        writer.Flush();

        string line = null;

        while (!string.IsNullOrEmpty(line =
reader.ReadLine()))
        {
            writer.WriteLine("Echoing back: {0}", line);
            writer.Flush();
        }
    }
}
```

Now we are able to send and receive data between the client and the server. It doesn't do anything useful yet, but we are getting there... An important thing to note is that you always want to flush the buffer out to the client after your writes. By default, when you write to the `StreamWriter`, it keeps it in a buffer locally. When you call `Flush`, it actually sends it to the client. To test, you can fire up the telnet application on the command line and connect to your server: *telnet localhost 21*. If you don't have telnet installed, you can install it from the command line with the following command on the command line in Windows 7: *pkgmgr /iu:"TelnetClient"*.

Format of commands and replies

Commands

Section 4.1 of the specification, FTP Commands, details what each command is and how each command should work.

> The commands begin with a command code followed by an argument field. The command codes are four or fewer alphabetic characters. Upper and lower case alphabetic characters are to be treated identically. The argument field consists of a variable length character string ending with the character sequence <CRLF>.

An example of an FTP command for sending the username to the server would be *USER my_user_name*. Section 5.3.1 defines the syntax for all commands.

Replies

Section 4.2, FTP Replies, details how the server should respond to each command.

> An FTP reply consists of a three digit number ... followed by some text. A reply is defined to contain the 3-digit code, followed by Space <SP>, followed by one line of text, and terminated by the Telnet end-of-line code.

There is more in the specification, including how to send multi-line replies, and I encourage the reader to read it and understand it, but we won't be covering that here. As an example of a single line reply, the server might send *200 Command OK*. The FTP client will know that the code 200 means success, and doesn't care about the text. The text is meant for the human on the other end. Section 5.4 defines all of the possible replies for each command.

What to do on connect?

Let's take a look at what the FTP specification says we should do when we get a connection. Section 5.4, Sequencing of Commands and Replies, details how the client and server should communicate. According to Section 5.4, under normal circumstances, a server will send a 220 reply when the connection is established. This lets the client know the server is ready to receive commands. From here on, I will use the excellent FTP client FileZilla for my testing. It is easy to use and shows the raw commands and replies in the top window, which is great for debugging. With that, let's modify the `HandleAcceptTcpClient` method:

```
TcpClient client = _listener.EndAcceptTcpClient(result);
```

```
_listener.BeginAcceptTcpClient(HandleAcceptTcpClient,
_listener);

NetworkStream stream = client.GetStream();

using (StreamWriter writer = new StreamWriter(stream,
Encoding.ASCII))
using (StreamReader reader = new StreamReader(stream,
Encoding.ASCII))
{
    writer.WriteLine("220 Ready!");
    writer.Flush();

    string line = null;

    while (!string.IsNullOrEmpty(line =
reader.ReadLine()))
    {
        Console.WriteLine(line);

        writer.WriteLine("502 I DON'T KNOW");
        writer.Flush();
    }
}
```

Now we are sending what the spec says to send, a 220 response on connect. Then we just go into a loop, listening for commands from the server. We haven't implemented any, so we just respond with *502 Command Not Implemented*. Looking at the specification, we can see that the number of commands is fairly large, and it would probably be a good idea at this point to split out handling the logic of a current connection to its own class. With that, let's create the `ClientConnection` class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.IO;
using System.Text;
using System.Net.Sockets;
using System.Net;
using log4net;
using System.Security.Cryptography.X509Certificates;
using System.Net.Security;

namespace SharpFtpServer
{
    public class ClientConnection
    {
        private TcpClient _controlClient;

        private NetworkStream _controlStream;
        private StreamReader _controlReader;
        private StreamWriter _controlWriter;

        private string _username;

        public ClientConnection(TcpClient client)
        {
            _controlClient = client;
```

```
            _controlStream = _controlClient.GetStream();

            _controlReader = new
StreamReader(_controlStream);
            _controlWriter = new
StreamWriter(_controlStream);
        }

        public void HandleClient(object obj)
        {
            _controlWriter.WriteLine("220 Service
Ready.");
            _controlWriter.Flush();

            string line;

            try
            {
                while (!string.IsNullOrEmpty(line =
_controlReader.ReadLine()))
                {
                    string response = null;

                    string[] command = line.Split(' ');

                    string cmd =
command[0].ToUpperInvariant();
                    string arguments = command.Length > 1
? line.Substring(command[0].Length + 1) : null;

                    if
(string.IsNullOrWhiteSpace(arguments))
                        arguments = null;

                    if (response == null)
                    {
                        switch (cmd)
                        {
                            case "USER":
                                response =
User(arguments);
                                break;
                            case "PASS":
                                response =
Password(arguments);
                                break;
                            case "CWD":
                                response =
ChangeWorkingDirectory(arguments);
                                break;
                            case "CDUP":
                                response =
ChangeWorkingDirectory("..");
                                break;
                            case "PWD":
                                response = "257 \"/\" is
current directory.";
                                break;
                            case "QUIT":
                                response = "221 Service
closing control connection";
                                break;

                            default:
                                response = "502 Command
```

```
not implemented";
                                    break;
                            }
                        }

                        if (_controlClient == null ||
!_controlClient.Connected)
                        {
                            break;
                        }
                        else
                        {

_controlWriter.WriteLine(response);
                            _controlWriter.Flush();

                            if (response.StartsWith("221"))
                            {
                                break;
                            }
                        }
                    }
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex);
                    throw;
                }
            }

            #region FTP Commands

            private string User(string username)
            {
                _username = username;

                return "331 Username ok, need password";
            }

            private string Password(string password)
            {
                if (true)
                {
                    return "230 User logged in";
                }
                else
                {
                    return "530 Not logged in";
                }
            }

            private string ChangeWorkingDirectory(string
pathname)
            {
                return "250 Changed to new directory";
            }

            #endregion
        }
}
```

As you can see, we have moved the code to handle the connection to the client into the `ClientConnection` class, and I have added a

few FTP commands, like USER, PASS, CWD, and CDUP. The constructor gets the `TcpClient` and opens the streams. Then we can call `HandleClient` to tell the client we are ready for commands and enter the command loop. The first thing we do in the command loop is split the line we received from the client on the SPACE character. The first item will be the command. Since the specification says casing does not matter for commands, we send it to uppercase to use in a `switch` statement. Using this class above, we can modify our `HandleAcceptTcpClient` method to the following:

```
private void HandleAcceptTcpClient(IAsyncResult result)
{
    _listener.BeginAcceptTcpClient(HandleAcceptTcpClient,
_listener);
    TcpClient client =
_listener.EndAcceptTcpClient(result);

    ClientConnection connection = new
ClientConnection(client);

    ThreadPool.QueueUserWorkItem(connection.HandleClient,
client);
}
```

`ThreadPool.QueueUserWorkItem` creates a new background thread the server will use to handle the request. This keeps the foreground thread free, and allows the .NET Framework to manage the threads for you.

With the new `ClientConnection` class and the modifications to `HandleAcceptTcpClient` above, we now have a partially functional FTP server! Just run your project and use FileZilla to connect. Use any username and password you want at the moment. As you can see, we always return true in the `Password` method (obviously this would need to change before considering this a real FTP server). Sample output from FileZilla with the above code:

```
Status: Connecting to 127.0.0.1:21...
Status: Connection established, waiting for welcome
message...
Response: 220 Service Ready.
Command: USER rick
Response: 331 Username ok, need password
Command: PASS ****
Response: 230 User logged in
Command: SYST
Response: 502 Command not implemented
Command: FEAT
Response: 502 Command not implemented
Status: Connected
Status: Retrieving directory listing...
Command: PWD
Response: 257 "/" is current directory.
Command: TYPE I
Response: 502 Command not implemented
Error: Failed to retrieve directory listing
```

As you can see above, when FileZilla first connects, the server sends a *220 Service Ready* response. Then the client sends the USER command with the argument of whatever username you specified. It keeps going all the way until it issues the TYPE command. Our server doesn't know how to execute that command yet, so let's start there.

The TYPE command

In Section 5.3.1, the TYPE command is declared as TYPE <SP> <type-code> <CRLF>. This means it is the TYPE command, followed by a space, followed by some type code, followed by the end-of-line characters. Section 5.3.2 defines the arguments for different commands. <type-code> is defined as being one of the following: I, A followed by an optional space and <form-code>, E followed by an optional space and <form-code>, or L followed by a required space and <byte-size>. The same section defines <form-code> to be one of the following: N, T, or C. Using this, we can define a method to handle this command:

```
private string Type(string typeCode, string formatControl)
{
    string response = "";

    switch (typeCode)
    {
        case "I":
            break;
        case "A":
            break;
        case "E":
            break;
        case "L":
            break;
        default:
            break;
    }

    switch (formatControl)
    {
        case "N":
            break;
        case "T":
            break;
        case "C":
            break;
    }

    return response;
}
```

We now have a model for our method, but what does it actually do? Section 4.1.2 defines the Transfer Parameter Commands, including the TYPE command. This is the way to transfer data, A = ASCII, I = Image, E = EBCDIC, and L = Local byte size. According to Section

5.1, the minimum implementation for an FTP server, only TYPE A is required, with a default format control of N (non-print). The rest we will signal to the client we can't support for now.

Now our `Type` method looks like the following:

```
private string Type(string typeCode, string formatControl)
{
    string response = "";

    switch (typeCode)
    {
        case "A":
            response = "200 OK";
            break;
        case "I":
        case "E":
        case "L":
        default:
            response = "504 Command not implemented for
that parameter.";
            break;
    }

    if (formatControl != null)
    {
        switch (formatControl)
        {
            case "N":
                response = "200 OK";
                break;
            case "T":
            case "C":
            default:
                response = "504 Command not implemented
for that parameter.";
                break;
        }
    }

    return response;
}
```

We need to add it to our command loop `switch` statement in the `HandleClient` method as well.

```
case "TYPE":
    string[] splitArgs = arguments.Split(' ');
    response = Type(splitArgs[0], splitArgs.Length > 1 ?
splitArgs[1] : null);
    break;
```

Now let's run FileZilla again...

```
Status: Connecting to 127.0.0.1:21...
Status: Connection established, waiting for welcome
message...
Response: 220 Service Ready.
Command: USER rick
Response: 331 Username ok, need password
```

```
Command: PASS ****
Response: 230 User logged in
Status: Connected
Status: Retrieving directory listing...
Command: PWD
Response: 257 "/" is current directory.
Command: TYPE I
Response: 504 Command not implemented for that parameter.
Error: Failed to retrieve directory listing
```

Looks like FileZilla isn't content with the bare minimum server implementation. It seems to require the Image type for transferring the directory listing. This means we now need to store what transfer type we are supposed to use on the data connection, which means a new class level variable. Now we can expand our Type method to look like this:

```
private string Type(string typeCode, string formatControl)
{
    string response = "500 ERROR";

    switch (typeCode)
    {
        case "A":
        case "I":
            _transferType = typeCode;
            response = "200 OK";
            break;
        case "E":
        case "L":
        default:
            response = "504 Command not implemented for
that parameter.";
            break;
    }

    if (formatControl != null)
    {
        switch (formatControl)
        {
            case "N":
                response = "200 OK";
                break;
            case "T":
            case "C":
            default:
                response = "504 Command not implemented
for that parameter.";
                break;
        }
    }

    return response;
}
```

Running FileZilla again gives us the following:

```
Status: Connecting to 127.0.0.1:21...
Status: Connection established, waiting for welcome
message...
Response: 220 Service Ready.
```

```
Command: USER rick
Response: 331 Username ok, need password
Command: PASS ****
Response: 230 User logged in
Status: Connected
Status: Retrieving directory listing...
Command: PWD
Response: 257 "/" is current directory.
Command: TYPE I
Response: 200 OK
Command: PASV
Response: 502 Command not implemented
Command: PORT 127,0,0,1,239,244
Response: 502 Command not implemented
Error: Failed to retrieve directory listing
```

We got further this time, but now it is trying to use the PASV and PORT commands, neither of which are implemented. Let's go back to the specification to see what they are... PASV or Passive, is a request to the server to open a new port the client can connect to for data transfer. This is useful when the client is behind a firewall; as the server won't be able to connect directly to a port on the client, the client has to initiate it. PORT on the other hand instructs the server to connect to the client on the given IP and port. Now we start getting into the other connection, the data connection...

The data connection

The data connection is used when the client or server needs to transfer files or other data, such as directory listings, that would be too large to send through the control connection. This connection does not always need to exist, and is typically destroyed when no longer needed. As stated above, there are two types of data connections, passive and active. With active, the server initiates a data connection to the client given the client's IP and port as arguments. With passive, the server begins listening on a new port, sends that port info back to the client in the response, and waits for the client to connect.

The PORT command

Let's start with the PORT command first. This command tells the server to connect to a given IP address and port. This is typically the IP address of the client, but it doesn't have to be. The client could send the server the IP address of another server, which would then accept the data connection for the client using the FXP or File eXchange Protocol. This is useful for server to server transfers, where the client doesn't need to receive the data itself. FXP will not be covered in this article. The PORT command takes as its argument a comma separated list of numbers. Each of these numbers represents

a single byte. An IPv4 address consists of four bytes (0 - 255). The port consists of a 16-bit integer (0 - 65535). The first thing we do is convert each of these numbers into a byte and store them in two separate arrays. The IP address in the first array, and the port in the second array. The IP address can be created directly from the byte array, but the port needs to be converted to an integer first. To do this, we take advantage of the `BitConverter` class. The specification says to send the high order bits first (most significant byte stored first), but depending on your CPU architecture, your CPU may be expecting the least significant byte first. These two differences in storing byte orders is called Endianness. Big Endian format stores the most significant first and is common on CPUs used in big iron, Little Endian stores the least significant byte first and is more common on desktop machines. To take this into consideration, we must reverse the array if the current architecture is Little Endian.

```
if (BitConverter.IsLittleEndian)
    Array.Reverse(port);
```

After reversing it if necessary, we can convert it directly.

```
BitConverter.ToInt16(port, 0)
```

Once we have identified the endpoint to connect to, we send back the 200 response. When the client tries to make use of the data connection, we will connect to that endpoint to send/receive data.

The PASV command

The PASV (passive) command tells the server to open a port to listen on, instead of connecting directly to the client. This is useful for situations where the client is behind a firewall, and cannot accept incoming connections. This command takes no arguments, but requires that the server respond with the IP address and port the client should connect to. In this case, we can create a new `TcpListener` for the data connection. In the constructor, we specify port 0, which tells the system we do not care which port is used. It will return an available port between 1024 and 5000.

```
IPAddress localAddress =
((IPEndPoint)_controlClient.Client.LocalEndPoint).Address;

_passiveListener = new TcpListener(localAddress, 0);
_passiveListener.Start();
```

Now that we are listening, we need to send the client back what IP address and port we are listening on.

```
IPEndPoint localEndpoint =
((IPEndPoint)_passiveListener.LocalEndpoint);

byte[] address = localEndpoint.Address.GetAddressBytes();
short port = (short)localEndpoint.Port;

byte[] portArray = BitConverter.GetBytes(port);

if (BitConverter.IsLittleEndian)
    Array.Reverse(portArray);

return string.Format("227 Entering Passive Mode ({0},{1},
{2},{3},{4},{5})",
              address[0], address[1], address[2],
address[3], portArray[0], portArray[1]);
```

As you can see, we are using the `BitConverter` class again and making sure we are checking for correct Endianness.

Adding data control commands to the command loop

Now that we have the `Passive` and `Port` methods created, we need to add them to our command loop.

```
case "PORT":
    response = Port(arguments);
    break;
case "PASV":
    response = Passive();
    break;
```

Now when we connect with FileZilla again, we get the following:

```
Status: Connecting to 127.0.0.1:21...
Status: Connection established, waiting for welcome
message...
Response: 220 Service Ready.
Command: USER rick
Response: 331 Username ok, need password
Command: PASS ****
Response: 230 User logged in
Command: SYST
Response: 502 Command not implemented
Command: FEAT
Response: 502 Command not implemented
Status: Connected
Status: Retrieving directory listing...
Command: PWD
Response: 257 "/" is current directory.
Command: TYPE I
Response: 200 OK
Command: PASV
Response: 227 Entering Passive Mode (127,0,0,1,197,64)
Command: LIST
Response: 502 Command not implemented
Error: Failed to retrieve directory listing
```

Looks like we need to implement the LIST command next...

The LIST command

The LIST command sends back a list of file system entries for the specified directory. If no directory is specified, we assume the current working directory. This is the first command we are implementing that will make use of the data connection we created above, so let's go through it step by step.

```
private string List(string pathname)
{
    if (pathname == null)
    {
        pathname = string.Empty;
    }

    pathname = new
DirectoryInfo(Path.Combine(_currentDirectory,
pathname)).FullName;

    if (IsPathValid(pathname))
    {
```

As you can see, the `List` method we are creating takes a single argument for the path. We do some quick verifications to make sure the path is valid, then...

```
if (_dataConnectionType == DataConnectionType.Active)
    {
        _dataClient = new TcpClient();
        _dataClient.BeginConnect(_dataEndpoint.Address,
_dataEndpoint.Port, DoList, pathname);
    }
    else
    {
        _passiveListener.BeginAcceptTcpClient(DoList,
pathname);
    }

    return string.Format("150 Opening {0} mode data
transfer for LIST", _dataConnectionType);
}

return "450 Requested file action not taken";
```

We check to see what type of data connection we are using. If we are using an Active connection, we need to initiate the connection to the client. If using Passive, we need to be ready to accept the connection. Both of these are done asynchronously, so that the main thread can return the 150 response back to the client. Both ways call the `DoList` method once they are connected. We pass the current path as a state object that can be retrieved in the `DoList` method.

The `DoList` method is actually going to do our heavy lifting here. We start by ending the connection attempt. We have to do it differently depending on which way we are connecting, Active or Passive. We also pull the path out from the state object.

```
private void DoList(IAsyncResult result)
{
    if (_dataConnectionType == DataConnectionType.Active)
    {
        _dataClient.EndConnect(result);
    }
    else
    {
        _dataClient =
_passiveListener.EndAcceptTcpClient(result);
    }

    string pathname = (string)result.AsyncState;
```

We now have reference to a TcpClient that we can use to send/receive data. The LIST command specifies to use ASCII or EBCDIC. In our case we are using ASCII, so we create a `StreamReader` and `StreamWriter` to make communication easier.

```
using (NetworkStream dataStream = _dataClient.GetStream())
{
    _dataReader = new StreamReader(dataStream,
Encoding.ASCII);
    _dataWriter = new StreamWriter(dataStream,
Encoding.ASCII);
```

Now that we have a way to write data back to the client, we need to output the list. Let's start with the directories. The specification does not dictate the format of the data, but is good to use what a UNIX system would output for *ls -l*. Here is an example of a format.

```
IEnumerable<string> directories =
Directory.EnumerateDirectories(pathname);

foreach (string dir in directories)
{
    DirectoryInfo d = new DirectoryInfo(dir);

    string date = d.LastWriteTime < DateTime.Now -
TimeSpan.FromDays(180) ?
        d.LastWriteTime.ToString("MMM dd  yyyy") :
        d.LastWriteTime.ToString("MMM dd HH:mm");

    string line = string.Format("drwxr-xr-x    2 2003
2003     {0,8} {1} {2}", "4096", date, d.Name);

    _dataWriter.WriteLine(line);
    _dataWriter.Flush();
}
```

As I said earlier, there is no set standard for formatting these lines. I have found that this format works well with FileZilla. You may have

different results with different clients. Above, we are just going through each directory in the current directory and writing a single line to the client describing each one. We do something similar for files...

```
IEnumerable<string> files =
Directory.EnumerateFiles(pathname);

foreach (string file in files)
{
    FileInfo f = new FileInfo(file);

    string date = f.LastWriteTime < DateTime.Now -
TimeSpan.FromDays(180) ?
        f.LastWriteTime.ToString("MMM dd  yyyy") :
        f.LastWriteTime.ToString("MMM dd HH:mm");

    string line = string.Format("-rw-r--r--    2 2003
2003     {0,8} {1} {2}", f.Length, date, f.Name);

    _dataWriter.WriteLine(line);
    _dataWriter.Flush();
}
```

Once we are done writing the list, we close the data connection and send a message to the client on the control connection, letting the client know the transfer is complete:

```
_dataClient.Close();
_dataClient = null;

_controlWriter.WriteLine("226 Transfer complete");
_controlWriter.Flush();
```

Now we add the LIST command to the command loop:

```
case "LIST":
    response = List(arguments);
    break;
```

And re-run FileZilla...

```
Status: Connecting to 127.0.0.1:21...
Status: Connection established, waiting for welcome
message...
Response: 220 Service Ready.
Command: USER rick
Response: 331 Username ok, need password
Command: PASS ****
Response: 230 User logged in
Status: Connected
Status: Retrieving directory listing...
Command: PWD
Response: 257 "/" is current directory.
Command: TYPE I
Response: 200 OK
Command: PASV
Response: 227 Entering Passive Mode (127,0,0,1,198,12)
```

```
Command: LIST
Response: 150 Opening Passive mode data transfer for LIST
Response: 226 Transfer complete
Status: Directory listing successful
```

You should see a listing of files/folders in the directory you specified. Hurray, something works! Now we need to implement some more functionality. Let's try downloading a file in FileZilla. You should get a result something like this:

```
Status: Connecting to 127.0.0.1:21...
Status: Connection established, waiting for welcome
message...
Response: 220 Service Ready.
Command: USER rick
Response: 331 Username ok, need password
Command: PASS ****
Response: 230 User logged in
Status: Connected
Status: Starting download of /somefile.txt
Command: CWD /
Response: 200 Changed to new directory
Command: TYPE A
Response: 200 OK
Command: PASV
Response: 227 Entering Passive Mode (127,0,0,1,198,67)
Command: RETR somefile.txt
Response: 502 Command not implemented
Error: Critical file transfer error
```

Looks like we now need the RETR command...

The RETR command

The RETR (retrieve) command is the command used to download a file from the server to the client. This is going to work similar to the LIST command, except we need to send the data differently depending on whether the client wants it transferred as ASCII or Image (binary).

```
private string Retrieve(string pathname)
{
    pathname = NormalizeFilename(pathname);

    if (IsPathValid(pathname))
    {
        if (File.Exists(pathname))
        {
            if (_dataConnectionType ==
DataConnectionType.Active)
            {
                _dataClient = new TcpClient();

_dataClient.BeginConnect(_dataEndpoint.Address,
_dataEndpoint.Port, DoRetrieve, pathname);
            }
```

```
            else
            {

_passiveListener.BeginAcceptTcpClient(DoRetrieve,
pathname);
            }

            return string.Format("150 Opening {0} mode
data transfer for RETR", _dataConnectionType);
        }
    }

    return "550 File Not Found";
}
```

As you can see above, we do some quick validation on the path we receive, then connect asynchronously depending on whether we are in Active or Passive mode. We pass off to the `DoRetrieve` method once we are connected.

Our `DoRetrieve` method should look very similar to the `DoList` method above.

```
private void DoRetrieve(IAsyncResult result)
{
    if (_dataConnectionType == DataConnectionType.Active)
    {
        _dataClient.EndConnect(result);
    }
    else
    {
        _dataClient =
_passiveListener.EndAcceptTcpClient(result);
    }

    string pathname = (string)result.AsyncState;

    using (NetworkStream dataStream =
_dataClient.GetStream())
    {
```

As you can see, the first half of the method is the same. Once we get the `NetworkStream`, we just need to transfer the file in the manner requested. First we open the file for reading...

```
using (FileStream fs = new FileStream(pathname,
FileMode.Open, FileAccess.Read))
{
```

Then copy the file stream to the data stream...

```
CopyStream(fs, dataStream);
```

Then close our data connection...

```
_dataClient.Close();
_dataClient = null;
```

And finally send a notice back to the client on the control connection...

```
    _controlWriter.WriteLine("226 Closing data connection,
file transfer successful");
    _controlWriter.Flush();
}
```

The `CopyStream` methods are detailed below:

```
private static long CopyStream(Stream input, Stream
output, int bufferSize)
{
    byte[] buffer = new byte[bufferSize];
    int count = 0;
    long total = 0;

    while ((count = input.Read(buffer, 0, buffer.Length))
> 0)
    {
        output.Write(buffer, 0, count);
        total += count;
    }

    return total;
}

private static long CopyStreamAscii(Stream input, Stream
output, int bufferSize)
{
    char[] buffer = new char[bufferSize];
    int count = 0;
    long total = 0;

    using (StreamReader rdr = new StreamReader(input))
    {
        using (StreamWriter wtr = new StreamWriter(output,
Encoding.ASCII))
        {
            while ((count = rdr.Read(buffer, 0,
buffer.Length)) > 0)
            {
                wtr.Write(buffer, 0, count);
                total += count;
            }
        }
    }

    return total;
}

private long CopyStream(Stream input, Stream output)
{
    if (_transferType == "I")
    {
        return CopyStream(input, output, 4096);
    }
    else
    {
        return CopyStreamAscii(input, output, 4096);
    }
}
```

As you can see, we copy the file differently depending on how the transfer should take place. We only support ASCII and Image at this point.

Next we add the RETR command to the command loop.

```
case "RETR":
    response = Retrieve(arguments);
    break;
```

At this point you should be able to download files from your server!

What's next?

From here, you should continue to go through the specification, implementing commands. I have implemented several more in the attached solution. There are plenty of areas that can be made much more generic, including our data connection methods. In order to make this a useable FTP server, we would also need to add in some real user account management, security for ensuring users stay in their own directories, etc. I have also implemented FTPS (FTP over SSL) for using an encrypted link from the client to the server. This is useful because by default, all commands are sent in plain text (including passwords). FTPS is defined in a different RFC than the original specification, RFC 2228. Next is a brief discussion on its implementation.

FTPS - FTP over SSL

The AUTH command

The AUTH command signals to the server that the client wants to communicate via a secure channel. I have only implemented the TLS auth mode. TLS, or Transport Layer Security, is the latest implementation of SSL. Our `Auth` method below is deceptively simple.

```
private string Auth(string authMode)
{
    if (authMode == "TLS")
    {
        return "234 Enabling TLS Connection";
    }
    else
    {
        return "504 Unrecognized AUTH mode";
    }
}
```

This is because the first thing we need to do is return to the client whether we can accept their request to use encryption. Then, below our `switch` statement in the command loop, we check again to see if we are using the AUTH command, and then encrypt the stream.

```
if (_controlClient != null && _controlClient.Connected)
{
    _controlWriter.WriteLine(response);
    _controlWriter.Flush();

    if (response.StartsWith("221"))
    {
        break;
    }

    if (cmd == "AUTH")
    {
        _cert = new X509Certificate("server.cer");
```

The first thing we do is load an X509 certificate from a file. This certificate was created using the *makecert.exe* command available in the Windows SDK. It should be available to you by running the Visual Studio Command Prompt. The MSDN documentation can walk you through how to create a certificate. Here is the command I used for creating my test certificate:

```
makecert -r -pe -n "CN=localhost" -ss my -sr localmachine
-sky
   exchange -sp "Microsoft RSA SChannel Cryptographic
Provider" -sy 12 server.cer
```

Next we create the encrypted stream, and authenticate to the client.

```
_sslStream = new SslStream(_controlStream);
_sslStream.AuthenticateAsServer(_cert);
```

Finally, we set our stream reader and writer to use the encrypted stream. As we read, the underlying `SslStream` will decrypt for us, and as we write, it will encrypt for us.

```
_controlReader = new StreamReader(_sslStream);
_controlWriter = new StreamWriter(_sslStream);
```

This implementation does not protect the data connection, only the command connection. As such, the files transferred are still susceptible to interception. RFC 2228 does provide ways to protect the data stream, but they are not implemented here.

IPv6

Adding IPv6 support is actually very simple to do. If you aren't familiar with IPv6, you should start reading. In the case of FTP, IPv6 support was added in RFC 2428. This specification defines two new commands, EPSV and EPRT. These correspond with the PASV and PORT commands we covered above.

Our first change has to be to allow the server to listen on an IPv6 address. To do this, we modify our FtpServer class to use a new constructor:

```
public FtpServer(IPAddress ipAddress, int port)
{
    _localEndPoint = new IPEndPoint(ipAddress, port);
}
```

We also modify the Start method:

```
_listener = new TcpListener(_localEndPoint);
```

Now, when we start our server, we can use:

```
FtpServer server = new FtpServer(IPAddress.IPv6Any, 21);
```

Lets start adding the new commands now.

The EPRT Command

The EPRT command is the Extended PORT command. This command accepts arguments that specify the type of internet protocol to use (1 for IPv4 and 2 for IPv6), the ip address to connect to, and the port. This command is also less complicated that the original PORT command, since the IP Addresses are given in their string representation, not their byte representation. This way, we can just use the IPAddress.Parse method to get our address.

```
private string EPort(string hostPort)
{
    _dataConnectionType = DataConnectionType.Active;

    char delimiter = hostPort[0];

    string[] rawSplit = hostPort.Split(new char[] {
delimiter }, StringSplitOptions.RemoveEmptyEntries);

    char ipType = rawSplit[0][0];

    string ipAddress = rawSplit[1];
    string port = rawSplit[2];

    _dataEndpoint = new
IPEndPoint(IPAddress.Parse(ipAddress), int.Parse(port));

    return "200 Data Connection Established";
}
```

The first character in the command is the delimiter. This is usually the pipe "|" character, but it could be different, so we just see what the client sent us. We then split the arguments on the delimiter provided. The first argument is the network protocol to use, 1 for IPv4, 2 for IPv6. The next argument is the IP Address to connect to, and the final argument is the port to connect to. It turns out we don't actually need the first argument, since the IPAddress.Parse method automatically detects the type of IP Address to use.

Now we can add our command to the command loop:

```
case "EPRT":
    response = EPort(arguments);
    break;
```

The EPSV Command

Once again, our new command is simpler than the original. Since the EPSV command returns the endpoint information in a string format, instead of byte format, we can simplify our method:

```
private string EPassive()
{
    _dataConnectionType = DataConnectionType.Passive;

    IPAddress localIp =
((IPEndPoint)_controlClient.Client.LocalEndPoint).Address;

    _passiveListener = new TcpListener(localIp, 0);
    _passiveListener.Start();

    IPEndPoint passiveListenerEndpoint =
(IPEndPoint)_passiveListener.LocalEndpoint;

    return string.Format("229 Entering Extended Passive
Mode (|||{0}|)", passiveListenerEndpoint.Port);
}
```

And then we add our command to the command loop:

```
 case "EPSV":
    response = EPassive();
    break;
```

Data Connection Error

As it stands now, we end up getting a cryptic error about "This protocol version is not supported." if we connect using the EPRT command. It turns out we have to specify the AddressFamily (IPv4 or IPv6) when creating our _dataClient. So we change our instances where we create the _dataClient from:

```
_dataClient = new TcpClient();
```

to:

```
_dataClient = new TcpClient(_dataEndpoint.AddressFamily);
```

Final thoughts...

There is plenty more to do to make this a full featured FTP server. I hope this article will serve as a good start for learning how to work with asynchronous methods, network streams, encryption, and dealing with a simple command loop. I have created a project hosted at GitHub so you can see changes to this as it moves forward. I don't really have any current plans to make this into a full featured FTP server, but if anyone would like development access to continue the project, please let me know.

History

First release.

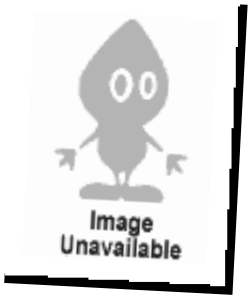
June 6th, 2012 - Added IPv6 Support for World IPv6 Day

October 7th, 2013 - Moved hosting from Google Code to GitHub

License

Share

About the Author



Rick Bassham
Software Developer (Senior)
United States

I have been a software developer since 2005, focusing on .Net applications with MS SQL backends, and recently, C++ applications in Linux, Mac OS X, and Windows.

Follow on Twitter Google

Comments and Discussions

48 messages have been posted for this article Visit **http://www.codeproject.com/Articles/380769/Creating-an-FTP-Server-in-Csharp-with-IPv-Support** to post and view comments on this article, or click **here** to get a print view with messages.