SYBEX Sample Chapter

C#™ Network Programming

Richard Blum

Chapter 7: Using the C# Sockets Helper Classes

Copyright © 2002 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

ISBN: 0-7821-4176-5

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the USA and other countries.

TRADEMARKS: Sybex has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer. Copyrights and trademarks of all products and services listed or described herein are property of their respective owners and companies. All rules and laws pertaining to said copyrights and trademarks are inferred.

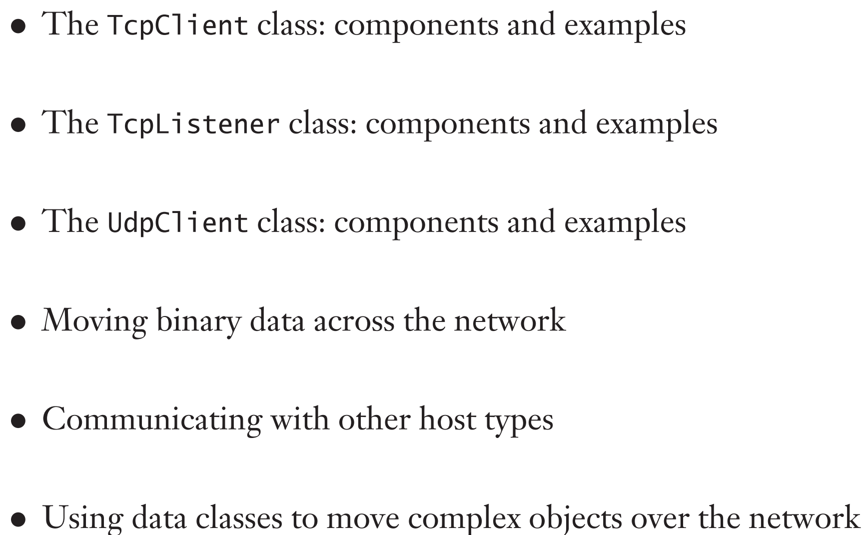
This document may contain images, text, trademarks, logos, and/or other material owned by third parties. All rights reserved. Such material may not be copied, distributed, transmitted, or stored without the express, prior, written consent of the owner.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturers. The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Sybex Inc.
1151 Marina Village Parkway
Alameda, CA 94501
U.S.A.
Phone: 510-523-8233
www.sybex.com

CHAPTER 7

Using The C# Sockets Helper Classes



- The `TcpClient` class: components and examples
- The `TcpListener` class: components and examples
- The `UdpClient` class: components and examples
- Moving binary data across the network
- Communicating with other host types
- Using data classes to move complex objects over the network

The two preceding chapters described how to use the low-level `Socket` class to create TCP and UDP network applications. While many hard-core network programmers are familiar with this type of programming, many with less experience find the `Socket` class difficult and confusing to use. To make network programming tasks easier for all of us, Microsoft provides a simplified set of helper classes to assist in creating full-featured network programs.

This chapter explains how to use the helper classes in network programs. You'll also see some real-world examples of how to use them. Further into the chapter, you'll see how to use the helper classes to move data between hosts on the network.

Often application data comes in many forms, some easy to transport and some not so easy. The chapter finishes up by showing a technique that can be used to transport complex binary data across the network using the helper classes.

The TcpClient Class

The `TcpClient` class, located in the `System.Net.Sockets` namespace, was designed to facilitate the writing of TCP client applications by bypassing the need to use the `Socket` class TCP functions. This section describes the `TcpClient` class and presents a simple TCP client application that uses the `TCPClient` class.

The TcpClient Class Constructors

The `TcpClient` class allows you to create a `TcpClient` object using three constructor formats:

TcpClient() The first constructor format creates a new `TcpClient` object, binding a socket to the local system address and a random TCP port. After the default `TcpClient` object is created, it must be connected to a remote device using the `Connect()` method, described in the upcoming section, "The TcpClient Class Methods." Here is the format for this constructor:

```
TcpClient newcon = new TcpClient();
newcon.Connect("www.ispnet.net", 8000);
```

TcpClient(IPEndPoint localEP) The second constructor allows you to specify a specific local IP address to use, as well as a specific TCP port number. This is most often used when the device has more than one network card, and you want to specifically send packets out a particular card. Again, after the `TcpClient` object is created, it must be connected to a remote device using the `Connect()` method. Here's the format:

```
IPEndPoint iep = new IPEndPoint(IPAddress,Parse("192.168.1.6"), 8000);
TcpClient newcon = new TcpClient(iep);
newcon.Connect("www.isp.net", 8000);
```

TcpClient(String host, int port) The third constructor format is the most frequently used. It allows you to specify the remote device within the constructor, and no `Connect()` method is needed. The remote device can be specified by its address and numeric TCP port value. The address can be either a string hostname, or a string IP address. The `TcpClient` constructor will automatically resolve the string hostname into the proper IP address. The format is as follows:

```
TcpClient newcon = new TcpClient("www.isp.net", 8000);
```

Once the `TcpClient` object has been created, several properties and methods are available with which to manipulate the object and transfer data back and forth with the remote device.

The TcpClient Class Methods

The `TcpClient` class contains a helpful collection of properties and methods to assist your efforts in writing TCP client applications. Table 7.1 shows the methods available for the `TcpClient` class.

TABLE 7.1: TcpClient Methods

Method	Description
`Close()`	Closes the TCP connection
`Connect()`	Attempts to establish a TCP connection with a remote device
`Equals()`	Determines if two `TcpClient` objects are equal
`GetHashCode()`	Gets a hash code suitable for use in hash functions
`GetStream()`	Gets a `Stream` object that can be used to send and receive data
`GetType()`	Gets the `Type` of the current instance
`ToString()`	Converts the current instance to a `String` object

As mentioned earlier, the `Connect()` method connects the `TcpClient` object to a remote device. Once the device is connected, the `GetStream()` object assigns a `NetworkStream` object to send and receive data:

```
TcpClient newcon = new TcpClient()
newcon.Connect("www.ispnet.net", 8000);
NetworkStream ns = new NetworkStream(newcon);
```

After the `NetworkStream` object is created, you can use the `Read()` and `Write()` methods to receive and send data (this is described in Chapter 5, "Connection-Oriented Sockets").

In addition to the `TcpClient` class methods, there are several properties that can be used with the `TcpClient` object, as described in Table 7.2. These properties allow you to set low-level `Socket` object options for the `TcpClient` object.

TABLE 7.2: TcpClient Object Properties

Property	Description
LingerState	Gets or sets the socket linger time
NoDelay	Gets or sets the delay time used for sending or receiving TCP buffers that are not full
ReceiveBufferSize	Gets or sets the size of the TCP receive buffer
ReceiveTimeout	Gets or sets the receive timeout value of the socket
SendBufferSize	Gets or sets the size of the TCP send buffer
SendTimeout	Gets or sets the send timeout value of the socket

Creating a Simple Client Program

The `TcpClient` class was designed to make network program creation as simple as possible. You have probably already concluded from the code snippets shown so far that a basic TCP client program can be written with just a few lines of code. Listing 7.1 is a simple TCP client program that will interact with Chapter 5's `SimpleTcpSrvr` program.

Listing 7.1 The TcpClientSample.cs program

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpClientSample
{
   public static void Main()
   {
      byte[] data = new byte[1024];
      string input, stringData;
      TcpClient server;

      try
      {
         server = new TcpClient("127.0.0.1", 9050);
      } catch (SocketException)
      {
         Console.WriteLine("Unable to connect to server");
         return;
      }
```

```
            NetworkStream ns = server.GetStream();

            int recv = ns.Read(data, 0, data.Length);
            stringData = Encoding.ASCII.GetString(data, 0, recv);
            Console.WriteLine(stringData);

            while(true)
            {
               input = Console.ReadLine();
               if (input == "exit")
                  break;
               ns.Write(Encoding.ASCII.GetBytes(input), 0, input.Length);
               ns.Flush();

               data = new byte[1024];
               recv = ns.Read(data, 0, data.Length);
               stringData = Encoding.ASCII.GetString(data, 0, recv);
               Console.WriteLine(stringData);
            }
            Console.WriteLine("Disconnecting from server...");
            ns.Close();
            server.Close();
         }
      }
```

Because the version of the `TcpClient` constructor used in this example automatically tries to connect to the specified remote server, it is a good idea to place it within a `try-catch` block in case the remote server is unavailable. As is true for the `Socket` method `Connect()` method, if the remote server is unavailable, a `SocketException` will be generated.

WARNING There is one catch to placing the `TcpClient` constructor within a `try-catch` block: variables instantiated in a `try-catch` block are only visible within the block. To make the `TcpClient` object visible outside the `try-catch` block, you must declare the variable outside the `try-catch` block, as shown in Listing 7.1 with the server `TcpClient` object.

After the `NetworkStream` object is created, you use the normal `Read()` and `Write()` methods to move the data:

```
while(true)
{
   input = Console.ReadLine();
   if (input == "exit")
      break;
   ns.Write(Encoding.ASCII.GetBytes(input), 0, input.Length);
```

```
      ns.Flush();

      data = new byte[1024];
      recv = ns.Read(data, 0, data.Length);
      stringData = Encoding.ASCII.GetString(data, 0, recv);
      Console.WriteLine(stringData);
   }
```

The `Read()` method requires three parameters:

- The data-byte array in which to place the received data
- The offset location in the buffer at which you want to start placing the data
- The length of the data buffer

Like the `Socket` method `Receive()`, the object's `Read()` method will read as much data as it can fit into its buffer. If the supplied buffer is too small, the leftover data stays in the stream for the next `Read()` method call.

The `Write()` method, too, requires three parameters:

- The data-byte array to send to the stream
- The offset location in the buffer from which you want to start sending data
- The length of the data to send

NOTE It is important to remember that TCP does not preserve message boundaries. This fact also applies to the `TcpClient` class. By now you'll recognize that this method should be handled the same way as the `Receive()` method in the TCP `Socket` class, creating a loop to ensure that all of the required data is read from the stream.

Testing the Program

You can test the new `TcpClientSample` program with the original `SimpleTcpSrvr` program (Listing 5.1) presented in Chapter 5. Just start `SimpleTcpSrvr` on the designated device, and run `TcpClientSample` from either a separate command-prompt window on the same machine or from a separate machine on the network. Remember to change the IP address in the program to the correct IP address for your server machine.

The `TcpClientSample` program should behave exactly like the `NetworkStreamTcpClient` program presented in Chapter 5 (Listing 5.9). You can type phrases at the console window and watch them displayed on the server and echoed back to the client.

Use the `windump` or `analyzer` commands if you want to watch the network traffic generated from your tests. Figure 7.1 shows a sample Analyzer output from monitoring a test.

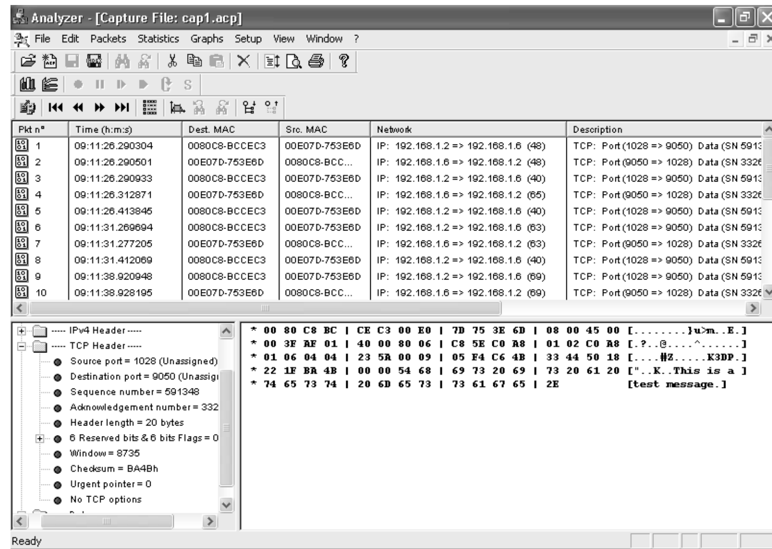


FIGURE 7.1: Sample Analyzer output from the TcpClientSample test

As you can see from the sample output, `TcpClientSample` behaves exactly the same as Chapter 5's `SimpleTcpClient` program, which used `Socket` objects. This demonstrates that you can often use the `TcpClient` class in place of `Socket` objects, saving yourself substantial programming effort.

The TcpListener Class

Like `TcpClient`, the `TcpListener` class (also located in the `System.Net.Sockets` namespace) provides a simplified way to create TCP server applications. This section describes the `TcpListener` class and shows how to use it in a simple TCP server application.

The TcpListener Class Constructors

The `TcpListener` class has three constructor formats:

TcpListener(int port) This constructor binds to a specific local port number.

TcpListener(IPEndPoint ie) This constructor binds to a specific local `EndPoint` object. TcpListener(IPAddress addr, int port)

TcpListener(IPAddress addr, int port) This constructor binds to a specific local `IPAddress` object and port number.

Unlike TcpClient, the TcpListener class constructor requires at least one parameter: the port number on which to listen for new connections. If the server machine has multiple network cards and you want to listen on a specific one, you can use an IPEndPoint object to specify the IP address of the desired card, along with the desired TCP port number to listen on. The constructor described last in the list just above allows you to specify the desired IP address using an IPAddress object, with the port number as a separate parameter.

The TcpListener Class Methods

The TcpListener class methods, listed in Table 7.3, are used to perform the necessary functions on the created TcpListener object.

TABLE 7.3: TcpListener Class Methods

Method	Description
AcceptSocket()	Accepts an incoming connection on the port and assign it to a Socket object
AcceptTcpClient()	Accepts an incoming connection on the port and assigns it to a TcpClient object
Equals()	Determines if two TcpListener objects are equal
GetHashCode()	Gets a hash code suitable for use in hash functions
GetType()	Gets the type of the current instance
Pending()	Determines if there are pending connection requests
Start()	Starts listening for connection attempts
Stop()	Stops listening for connection attempts (closes the socket)
ToString()	Creates a string representation of the TcpListener object

The procedure to create a TcpListener object and listen for incoming connections goes like this:

```
TcpListener server = new TcpListener(IPAddress.Parse("127.0.0.1"), 9050);
server.Start();
TcpClient newclient = server.AcceptTcpClient();
```

The Start() method is similar to the combination of Bind() and Listen() used in the Socket class. Start() binds the socket to the endpoint defined in the TcpListener constructor and places the TCP port in listen mode, ready to accept new connections. The AcceptTcpClient() method is comparable to the Accept() socket method, accepting incoming connection attempts and assigning them to a TcpClient object.

After the TcpClient object is created, all communication with the remote device is performed with the new TcpClient object rather than the original TcpListener object. The TcpListener

object can thus be used to accept other connections and pass them to other `TcpClient` objects. To close the `TcpListener` object, you must use the `Stop()` method:

```
server.Stop();
```

NOTE If you have any open client connections, you do not have to close them before the original `TcpListener` object is closed. However, you do have to remember to close the individual `TcpClient` objects using the `Close()` method.

A Simple Server Program

Now let's look at a simple example of a TCP server using the `TcpListener` class. This example, `TcpListenerSample.cs` in Listing 7.2, mimics the functionality of the original `SimpleTcpSrvr` program presented in Chapter 5 (Listing 5.1).

Listing 7.2 The TcpListenerSample.cs program

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpListenerSample
{
   public static void Main()
   {
      int recv;
      byte[] data = new byte[1024];

      TcpListener newsock = new TcpListener(9050);
      newsock.Start();
      Console.WriteLine("Waiting for a client...");

      TcpClient client = newsock.AcceptTcpClient();
      NetworkStream ns = client.GetStream();

      string welcome = "Welcome to my test server";
      data = Encoding.ASCII.GetBytes(welcome);
      ns.Write(data, 0, data.Length);

      while(true)
      {
         data = new byte[1024];
         recv = ns.Read(data, 0, data.Length);
         if (recv == 0)
            break;

         Console.WriteLine(
```

```
                Encoding.ASCII.GetString(data, 0, recv));
         ns.Write(data, 0, recv);
      }
      ns.Close();
      client.Close();
      newsock.Stop();
   }
}
```

The TcpListenerSample program first creates a TcpListener object, using a UDP port of 9050, and the Start() method places the new object in a listening mode. Then the AcceptTcpClient() method waits for an incoming TCP connection and requests and assigns it to a TcpClient object:

```
TcpListener newsock = new TcpListener(9050);
newsock.Start();

TcpClient client = newsock.AcceptTcpClient();
NetworkStream ns = client.GetStream();
```

With the TcpClient object established, a NetworkStream object is assigned to it to communicate with the remote client. All communication is done using the NetworkStream object's Read() and Write() methods. Remember to place all data into a byte array for the Write() method. Likewise, all received data from the Read() method must also be assigned to a byte array.

You can test the TcpListenerSample program by starting it up and connecting to it with the TcpClientSample program from the preceding section. They should behave together exactly like the SimpleTcpSrvr (Listing 5.1) and SimpleTcpClient (Listing 5.2) programs from Chapter 5. As usual, if you are testing these programs across a network, you can use the WinDump and Analyzer programs to watch the network traffic generated by each program.

Incorporating the StreamReader and StreamWriter Classes

Because the NetworkStream object uses streams to transfer data among the network hosts, you will have to handle the usual problems of identifying messages in the stream. Use the standard techniques outlined in Chapter 5 to delimit messages within the stream:

- Send fixed-size messages
- Send the message size before the message
- Use message delimiter characters

After it's created from the TcpClient object, you can use the NetworkStream object to create StreamReader and StreamWriter objects. (This is demonstrated in the "Using C# Streams with

TCP" section of Chapter 5.) These objects will automatically create message delimiters to simplify moving text across the network:

```
TcpClient client = new TcpClient("127.0.0.1", 9050);
NetworkStream ns = client.GetStream();
StreamReader sr = new StreamReader(ns);
StreamWriter sw = new StreamWriter(ns);

sw.WriteLine("This is a test");
sw.Flush();
string data = sr.ReadLine();
```

The `StreamReader` and `StreamWriter` classes by default use a line feed to delimit messages, which makes it a snap to distinguish[messages in TCP communications. Because the `ReadLine()` and `WriteLine()` methods both use `String` objects, text messages can be created and sent using the `String` class objects instead of your having to mess with the bulky data-byte arrays.

The UdpClient Class

The `UdpClient` class was created to help make UDP network programs simpler for network programmers. This section describes the `UdpClient` class and its methods and walks you through creating a simple UDP server and client program using the helper class.

The UdpClient Class Constructors

The `UdpClient` class has four formats of constructors:

UdpClient() This format creates a new `UdpClient` instance not bound to any specific address or port.

UdpClient(int port) This constructor binds the new `UdpClient` object to a specific UDP port number.

UdpClient(IPEndPoint iep) This constructor binds the new `UdpClient` object to a specific local IP address and port number.

UdpClient(string host, int port) This format binds the new `UdpClient` object to any local IP address and port and associates it with a specific remote IP address and port.

The `UdpClient` constructors work like their comparable `TcpClient` constructors. You can either let the system choose a UDP port for the application, or you can select a specific port in the constructor. If your UDP application must accept data on a specific port, you must define that port in the `UdpClient` constructor.

Once the `UdpClient` object is created, you can manipulate the underlying socket and move data using the various methods available.

The UdpClient Class Methods

The methods of the `UdpClient` class provide various functionality for controlling and moving data into and out of the UDP socket. Table 7.4 describes these methods.

TABLE 7.4: The UdpClient Class Methods

Method	Description
`Close()`	Closes the underlying socket
`Connect()`	Allows you to specify a remote IP endpoint to send and receive data with
`DropMulticastGroup()`	Removes the socket from a UDP multicast group
`Equals()`	Determines if two `UdpClient` objects are equal
`GetHashCode()`	Gets a hash code for the `UdpClient` object
`GetType()`	Gets the `Type` of the current object
`JoinMulticastGroup()`	Adds the socket to a UDP multicast group
`Receive()`	Receives data from the socket
`Send()`	Sends data to a remote host from the socket
`ToString()`	Creates a string representation of the `UdpClient` object

NOTE The `JoinMulticastGroup()` and `DropMulticastGroup()` methods allow you to program UDP applications to use IP multicasting. This feature is discussed in Chapter 10, "IP Multicasting."

Using the UdpClient Class in Programs

There are a few subtle differences between the `UdpClient` class's `Receive()` and `Send()` methods that make them different from the `Socket` methods `ReceiveFrom()` and `SendTo()`.

The Receive() Method

The `UdpClient` class uses the `Receive()` method to accept UDP packets on the specified interface and port. There is only one `Receive()` method format:

```
byte[] Receive(ref IPEndPoint iep)
```

The `Receive()` method accepts UDP packets on the IP address and UDP port specified by the `UdpClient` constructor, either system-specified values, or values set in the constructor.

Let's take a look at how the `Receive()` method format differs from the `ReceiveFrom()` method used with standard UDP `Socket` objects.

For starters, the data received from the socket is not placed in a byte array within the method call. It is returned by the method. You must specify an empty byte array for the received data.

The second difference between the `UdpClient` method `Receive()` and the `Socket` method `ReceiveFrom()`is the way the remote host information is returned. `ReceiveFrom()`places the remote host information in an `EndPoint` object, whereas `Receive()`uses an `IPEndPoint` object. This makes extracting the IP address and UDP port number of the remote host a little easier for the programmer.

The following code snippet demonstrates how to use the `Receive()` method in a UDP application:

```
IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
UdpClient newsock = new UdpClient(ipep);
byte[] data = new byte[1024];
IPEndPoint ipep2 = new IPEndPoint(IPAddress.Any, 0);
data = host.Receive(ref ipep2);
Console.WriteLine("The remote host is: {0}, port {1}",
    ipep2.Address, ipep2.Port);
Console.WriteLine(Encoding.ASCII.GetString(data));
```

In this code snippet, a UDP packet is accepted on UDP port 9050 from any network interface on the machine and is displayed on the console.

NOTE One nice feature of the `Receive()` method is what happens when more data is received than the buffer size specified can accommodate. Instead of throwing a `SocketException`, as the `Socket` object does, the `UdpClient` returns a data buffer large enough to handle the received data. The result: a handy feature that can save you lots of extra programming effort.

The Send() Method

The `Send()` method has three formats that can send data to a remote host:

Send(byte[] *data*, int *sz*) This format sends the byte array *`data`* of size *`sz`* to the default remote host. To use this format, you must specify a default remote UDP host using either `UdpClient` constructor, or the `Connect()` method:

```
UdpClient host = new UdpClient("127.0.0.1", 9050);
```

Send(byte[] *data*, int *sz*, IPEndPoint *iep*) This format sends the byte array *`data`* of size *`sz`* to the remote host specified by *`iep`*.

Send(byte[] *data*, int *sz*, string *host*, int *port*) This format sends the byte array `data` of size `sz` to the host `host` at port `port`.

TIP

If you are writing a UDP application that does not need to listen for incoming packets on a specific UDP port, you can use the `UdpClient` constructor that specifies the remote host information and then use the `Receive()` and `Send()` methods to move data back and forth with the remote host.

A Simple UdpClient Server Program

The `UdpSrvrSample.cs` program, shown in Listing 7.3, demonstrates using the `UdpClient` class methods in a server application environment.

Listing 7.3 The UdpSrvrSample.cs program

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class UdpSrvrSample
{
   public static void Main()
   {
      byte[] data = new byte[1024];
      IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
      UdpClient newsock = new UdpClient(ipep);

      Console.WriteLine("Waiting for a client...");

      IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);

      data = newsock.Receive(ref sender);

      Console.WriteLine("Message received from {0}:", sender.ToString());
      Console.WriteLine(Encoding.ASCII.GetString(data, 0, data.Length));

      string welcome = "Welcome to my test server";
      data = Encoding.ASCII.GetBytes(welcome);
      newsock.Send(data, data.Length, sender);

      while(true)
      {
         data = newsock.Receive(ref sender);

         Console.WriteLine(Encoding.ASCII.GetString(data, 0, data.Length));
         newsock.Send(data, data.Length, sender);
```

```
        }
    }
}
```

The `UdpSrvrSample` program creates a `UdpClient` object from an `IPEndPoint` object, specifying any IP address on the server and using UDP port 9050. The program immediately waits for an incoming UDP packet from any remote client, using the `Receive()` method:

```
IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
data = newsock.Receive(ref sender);
```

The sender variable stores the IP information of the client, which is then used for sending messages back to the client:

```
newsock.Send(data, data.Length, sender);
```

Because the `UdpClient` object does not know the IP information of the destination host, you must specify it for each `Send()` method call.

NOTE As seen in the `UdpSrvrSample` program, the data-byte array does not need to be reset to its full length after every `Receive()` method. This is a handy feature of the `UdpClient` class.

A Simple UdpClient Client Program

Here is the matching client program, `UdpClientSample.cs` (Listing 7.4), demonstrating how to use the `UdpClient` class methods in a UDP client application.

Listing 7.4 The UdpClientSample.cs program

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class UdpClientSample
{
   public static void Main()
   {
      byte[] data = new byte[1024];
      string input, stringData;
      UdpClient server = new UdpClient("127.0.0.1", 9050);

      IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);

      string welcome = "Hello, are you there?";
      data = Encoding.ASCII.GetBytes(welcome);
      server.Send(data, data.Length);
```

```
         data = server.Receive(ref sender);

         Console.WriteLine("Message received from {0}:", sender.ToString());
         stringData = Encoding.ASCII.GetString(data, 0, data.Length);
         Console.WriteLine(stringData);

         while(true)
         {
            input = Console.ReadLine();
            if (input == "exit")
               break;

            server.Send(Encoding.ASCII.GetBytes(input), input.Length);
            data = server.Receive(ref sender);
            stringData = Encoding.ASCII.GetString(data, 0, data.Length);
            Console.WriteLine(stringData);
         }
         Console.WriteLine("Stopping client");
         server.Close();
      }
   }
```

Because the `UdpClientSample` program does not need to listen to a specific UDP port for incoming messages, you can use the all-in-one constructor format, specifying the IP address and UDP port number of the remote host:

```
UdpClient server = new UdpClient("127.0.0.1", 9050);
```

Of course, if you are connecting to a remote UDP server, remember to place either the IP address or hostname of the remote server in the constructor instead of the loopback address.

After the `UdpClient` object is created, the `Send()` method sends the greeting message out to the server:

```
server.Send(data, data.Length);
```

Notice that because the destination host address information was already specified in the `UdpClient` constructor, you do not need to specify it in the `Send()` method.

Testing the Sample Programs

After compiling the `UdpSrvrSample` and `UdpClientSample` programs, you can test them on the same machine or on separate machines on the network.

The output looks exactly like the output from the `SimpleUdpClient` program (Listing 6.2) shown in Chapter 6, "Connectionless Sockets." Each message sent using the `Send()` method call is sent as a single UDP packet to the remote machine and read using a single `Receive()` method call.

Because the `UdpClient` class employs UDP packets to transmit messages, it suffers from one of the same problems as the UDP `Socket` objects, as described in Chapter 6. Specifically,

there is always the possibility that sent messages will not make it to the destination device, so you must compensate for that in your `UdpClient` programs. This is usually accomplished by using the retry techniques demonstrated in Chapter 6.

WARNING One problem with UDP `Socket` objects that is *not* found in `UdpClient` objects is lost data. If the data buffer supplied to the `UdpClient` method `Receive()` is too small for the incoming data, the buffer is returned to match the size of the data. No data is lost, and no `SocketExceptions` are thrown.

Moving Data across the Network

So far in this chapter you have seen how to move text messages efficiently from one device to another device across the network using the `Socket` helper classes. Certainly, for many applications this is all that is necessary; for many other applications, however, more advanced functionality is required to handle classes of data other than text such as binary data and groups of more than one type of data.

This section shows techniques for moving different types of binary data across the network. When programming to communicate with various systems, it is important that you understand how binary data is stored on a device and how it is transmitted across the network. This section also covers how to move complex datatypes, such as data elements in classes, among devices on the network.

Moving Binary Data

Whether you use TCP or UDP, sending binary data between two devices on a network is a complex topic. There are many possibilities for errors, and you must take them all into account. This section offers several suggestions for creating programs that move binary data successfully from one device to another.

Binary Data Representation

Perhaps the major issue when moving binary datatypes on a network is how they are represented. The various types of machines all represent binary datatype in their own way. You must ensure that the binary value on one machine turns out to be the same binary value on another machine.

Machines running a Microsoft Windows OS on an Intel (or compatible) processor platform store binary information using a set pattern for each datatype. It is important that you understand how this information is represented when sending binary data to a non-Windows remote host. Table 7.5 lists the binary datatypes used in C#.

TABLE 7.5: C# Binary Datatypes

Datatype	Bytes	Description
`sbyte`	1	Signed byte integer with values from –128 to 127
`byte`	1	Unsigned integer with values from 0 to 255
`short`	2	Signed short integer with values from –32,768 to 32,767
`ushort`	2	Unsigned short integer with values from 0 to 65,535
`int`	4	Standard integer with values from –2,147,483,648 to 2,147,483,647
`uint`	4	Unsigned integer with values from 0 to 4,294,967,295
`long`	8	Large signed integer with values from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`ulong`	8	Large unsigned integer with values from 0 to 18,446,744,073,709,551,615
`float`	4	A floating-point decimal number with values from 1.5×10^{-45} to 3.4 x 1,038, using 7-digit precision
`double`	8	A floating-point decimal number with values from 5.0×10^{-324} to 1.7×10^{308}, using 15–16-digit precision
`decimal`	16	A high-precision floating-point number with values from 1.0×10^{-28} to 7.9×10^{28}, using 28–29-digit precision

Each binary datatype must be converted into a raw byte array before it can be sent using the `Send()` or `SendTo()` methods. Fortunately, the .NET library provides a class specifically for this job: the `BitConverter` class.

Converting Binary Datatypes

As seen in the `VarTcpClient` (Listing 5.8) and `VarTcpSrvr` (Listing 5.7) programs in Chapter 5, the .NET System class supplies the `BitConverter` class to convert binary datatypes into byte arrays, and vice versa. This class is crucial to accurately sending binary datatypes across the network to remote hosts.

Sending Binary Datatypes

The `BitConverter` method `GetBytes()`converts a standard binary datatype to a byte array. For example:

```
int test = 1990;
byte[] data = BitConverter.GetBytes(test);
newsock.Send(data, data.Length);
```

This simple code snippet shows the conversion of a standard 4-byte integer value into a 4-byte byte array, which is then used in a `Write()` or `Send()` method call to forward the value to a remote device.

WARNING If you are sending binary datatypes using TCP, you cannot use the `StreamWriter` or `StreamReader` classes because they expect data to be sent in strings, not binary. Any nulls in the binary data will damage the string conversion.

NOTE When creating the byte array for the binary datatype, it is important that you allocate enough space in the byte array to contain all of the bytes of the binary datatype. If not all of the bytes of the converted binary datatype are sent, the receiving program will not be able to “reassemble” them back into the original datatype.

Receiving Binary Datatypes

As just stated in the preceding Note, the receiving program must be able to receive the byte array containing the binary datatype and convert it back into the original binary datatype. This is also done using `BitConverter` class methods. Table 7.6 lists the `Bitconverter` class’s methods for converting raw byte arrays into binary datatypes.

TABLE 7.6: BitConverter Methods for Converting Data

Method	Description
`ToBoolean()`	Converts a 1-byte byte array to a Boolean value
`ToChar()`	Converts a 2-byte byte array to a Unicode character value
`ToDouble()`	Converts an 8-byte byte array to a double floating-point value
`ToInt16()`	Converts a 2-byte byte array to a 16-bit signed integer value
`ToInt32()`	Converts a 4-byte byte array to a 32-bit signed integer value
`ToSingle()`	Converts a 4-byte byte array to a single-precision floating-point value
`ToString()`	Converts all bytes in the byte array to a string that represents the hexadecimal values of the binary data
`ToUInt16()`	Converts a 2-byte byte array to a 16-bit unsigned integer value
`ToUInt32()`	Converts a 4-byte byte array to a 32-bit unsigned integer value
`ToUing64()`	Converts an 8-byte byte array to a 64-bit unsigned integer value

All the converter methods have the same format:

```
BitConverter.ToInt16(byte[] data, int offset)
```

The byte array is the first parameter, and the second parameter is the offset location within the array where the conversion is to start. Note that the methods know how many bytes to use within the array to create the appropriate binary datatype.

Once the received byte array is converted to a binary datatype, you can use it in the program as any other value of that datatype, as shown here:

```
double total = 0.0;
byte[] data = newsock.Receive(ref sender);
double test = BitConverter.ToDouble(data, 0);
total += test;
```

Sample Programs

Sending binary information using UDP packets is fairly easy, assuming that you are sending one value per message. (We'll look at multiple-value situations in a later section.) Because UDP preserves message boundaries, you are guaranteed that if the packet arrives, there is only one data value in it. Assuming the server knows what type of binary data is in the packet, it is a snap to decode the value within the message back to its original binary value.

Listing 7.5 shows a sample UDP server program that reads binary data from the network.

Listing 7.5 The BinaryUdpSrvr.cs program

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class BinaryUdpSrvr
{
   public static void Main()
   {
      byte[] data = new byte[1024];
      IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
      UdpClient newsock = new UdpClient(ipep);

      Console.WriteLine("Waiting for a client...");

      IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);

      data = newsock.Receive(ref sender);

      Console.WriteLine("Message received from {0}:", sender.ToString());
      Console.WriteLine(Encoding.ASCII.GetString(data, 0, data.Length));

      string welcome = "Welcome to my test server";
      data = Encoding.ASCII.GetBytes(welcome);
      newsock.Send(data, data.Length, sender);

      byte[] data1 = newsock.Receive(ref sender);
      int test1 = BitConverter.ToInt32(data1, 0);
      Console.WriteLine("test1 = {0}", test1);

      byte[] data2 = newsock.Receive(ref sender);
```

```
        double test2 = BitConverter.ToDouble(data2, 0);
        Console.WriteLine("test2 = {0}", test2);

        byte[] data3 = newsock.Receive(ref sender);
        int test3 = BitConverter.ToInt32(data3, 0);
        Console.WriteLine("test3 = {0}", test3);

        byte[] data4 = newsock.Receive(ref sender);
        bool test4 = BitConverter.ToBoolean(data4, 0);
        Console.WriteLine("test4 = {0}", test4.ToString());

        byte[] data5 = newsock.Receive(ref sender);
        string test5 = Encoding.ASCII.GetString(data5);
        Console.WriteLine("test5 = {0}", test5);

        newsock.Close();
    }
}
```

The `BinaryUdpSrvr` program uses the `Receive()` method to wait for a remote client to send a greeting message, then returns its welcome banner to the client.

Next, it expects to receive five test messages in a row from the remote client. Each message contains a different type of data. The order in which the messages appear is critical, because each message is decoded back into the appropriate datatype based on its position in the receipt order. Of course, with UDP, there's no guarantee that the packets will arrive at all, so this is not a good real-world example. You must be careful when using this technique with UDP messages.

One important thing to note about this program is that the `BitConverter` class contains methods to convert raw binary data into binary datatypes, but not into a text datatype. The `BitConverter` method `ToString()`does exist, but its role is different from what you probably expect. Rather than converting the raw binary data into a printable string, it converts the raw data into a string representation of the binary data in hexadecimal. For example, this code snippet:

```
string data = "this is a test";
string test = BitConverter.ToString(Encoding.ASCII.GetBytes(data));
Console.WriteLine("data = '{0}'", data);
Console.WriteLine("test = '{0}'", test);
```

produces these results:

```
C:\>test
data = 'this is a test'
test = '74-68-69-73-20-69-73-20-61-20-74-65-73-74'

C:\>
```

NOTE The BitConverter method ToString()is a handy way to display the hexadecimal values of a byte array, but if you want to display the actual converted text string, you must use the Encoding.ASCII.GetString() method, as shown in the preceding BinaryUdpSrvr program (Listing 7.5).

The BinaryUdpClient program, Listing 7.6, is the counterpart to the BinaryUdpSrvr program. Here it sends five types of data to the server program.

Listing 7.6 The BinaryUdpClient.cs program

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class BinaryUdpClient
{
   public static void Main()
   {
      byte[] data = new byte[1024];
      string stringData;
      UdpClient server = new UdpClient("127.0.0.1", 9050);

      IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);

      string welcome = "Hello, are you there?";
      data = Encoding.ASCII.GetBytes(welcome);
      server.Send(data, data.Length);

      data = new byte[1024];
      data = server.Receive(ref sender);

      Console.WriteLine("Message received from {0}:", sender.ToString());
      stringData = Encoding.ASCII.GetString(data, 0, data.Length);
      Console.WriteLine(stringData);

      int test1 = 45;
      double test2 = 3.14159;
      int test3 = -1234567890;
      bool test4 = false;
      string test5 = "This is a test.";

      byte[] data1 = BitConverter.GetBytes(test1);
      server.Send(data1, data1.Length);

      byte[] data2 = BitConverter.GetBytes(test2);
```

```
        server.Send(data2, data2.Length);

        byte[] data3 = BitConverter.GetBytes(test3);
        server.Send(data3, data3.Length);

        byte[] data4 = BitConverter.GetBytes(test4);
        server.Send(data4, data4.Length);

        byte[] data5 = Encoding.ASCII.GetBytes(test5);
        server.Send(data5, data5.Length);

        Console.WriteLine("Stopping client");
        server.Close();
    }
}
```

BinaryUdpClient uses the Send() method to send a greeting banner to the server (specified by an address in the UdpClient constructor) and waits for the welcome banner to be returned. Once the welcome banner is received, the program sends a series of five messages, each containing data in a different datatype.

The output from the BinaryUdpSrvr program should look like this:

```
C:\>BinaryUdpSrvr
Waiting for a client...
Message received from 127.0.0.1:1252:
Hello, are you there?
test1 = 45
test2 = 3.14159
test3 = -1234567890
test4 = False
test5 = This is a test.

C:\>
```

Each of the binary datatypes was successfully transmitted to the server program across the network and can be used in other calculations within the server program if necessary.

WinDump and Analyzer, as usual, will let you watch the actual data packets if you are testing these programs across a network.

Note how each datatype is sent within the individual packets. The packet for the integer value contains the byte array for the integer value 45. The binary data is sent as the 4-byte value 0x2D 0x00 0x00 0x00, seen in the data section of the UDP packet.

Note the order in which the binary data value is sent in the packet. This representation format is an important feature of sending binary data that is discussed in the next section.

Communicating with Other Host Types

When sending binary datatypes between two devices that are both running a Microsoft Windows OS on an Intel microprocessor platform, you do not have to worry about how the binary data is represented. Each side of the network communications channel recognizes the binary data. The byte array produced from the `BitConverter.GetBytes()` method is converted to the proper binary datatype for the other machine using the `BitConverter.ToInt32()` method.

Of course, that's not the end of the story. The C# language in your network programs is being ported to other operating systems running on other CPU platforms. So it is possible and entirely likely that the binary datatype representations of the client and server programs may not be the same. This section describes how to meet this challenge and make your C# network programs ready to accommodate the formats of various platforms.

Binary Datatype Representation

The problem of dueling binary datatypes arises from the fact that CPU platforms may store binary datatypes differently. Because multiple bytes are used for the datatype, they can be stored one of two ways:

- The least significant byte first (called *little endian*)
- The most significant byte first (called *big endian*)

It is imperative that the binary datatype is interpreted correctly on each system, sending and receiving. If the wrong datatype representation is used to convert a raw binary byte array, your programs will be working with incorrect data.

Listing 7.7 is the `BinaryDataTest.cs` program, which uses the `BitConverter.ToString()` method to demonstrate how the different binary datatypes are stored on your system.

Listing 7.7 The BinaryDataTest.cs program

```
using System;
using System.Net;
using System.Text;

class BinaryDataTest
{
   public static void Main()
   {
      int test1 = 45;
      double test2 = 3.14159;
      int test3 = -1234567890;
      bool test4 = false;
      byte[] data = new byte[1024];
```

```
         string output;

         data = BitConverter.GetBytes(test1);
         output = BitConverter.ToString(data);
         Console.WriteLine("test1 = {0}, string = {1}", test1, output);

         data = BitConverter.GetBytes(test2);
         output = BitConverter.ToString(data);
         Console.WriteLine("test2 = {0}, string = {1}", test2, output);

         data = BitConverter.GetBytes(test3);
         output = BitConverter.ToString(data);
         Console.WriteLine("test3 = {0}, string = {1}", test3, output);

         data = BitConverter.GetBytes(test4);
         output = BitConverter.ToString(data);
         Console.WriteLine("test4 = {0}, string = {1}", test4, output);
      }
   }
```

All that happens here is that `BinaryDataTest` does some simple `BitConverter` operations on various datatypes, and it uses the `BitConverter ToString()` method to display the resulting byte array values. The output from the `BinaryDataTest` program should look like this:

```
C:\>BinaryDataTest
test1 = 45, string = 2D-00-00-00
test2 = 3.14159, string = 6E-86-1B-F0-F9-21-09-40
test3 = -1234567890, string = 2E-FD-69-B6
test4 = False, string = 00

C:\>
```

By looking at the simple integer value (test1) you can see that the standard byte representation used on this machine is little endian (the 2D value comes before the zeros). If this were a big endian system, the integer would be stored as 00-00-00-2D instead. So when sending data to a host that uses big endian data representation, errors will occur unless your program adjusts.

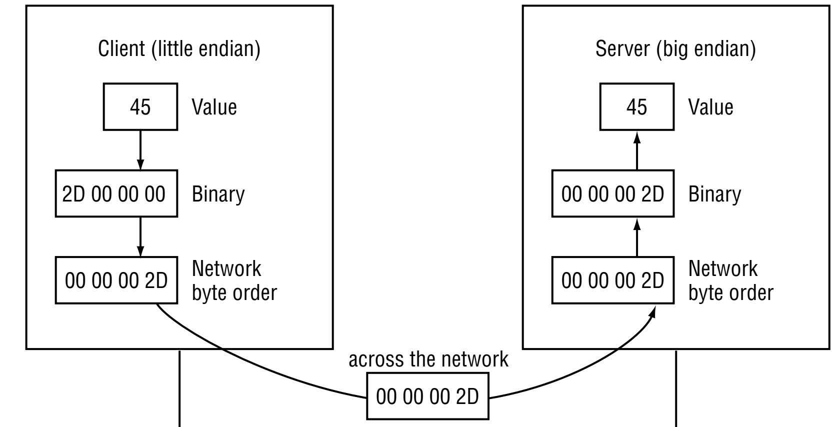
Converting Binary Data Representation

The problem of using different binary datatype representations is a significant issue in the Unix environment. Because so many platforms run Unix, you can never assume that the remote system will be using the same representation as yours. The Unix world has devised a solution: sending binary datatypes in a *generic* method.

The *network byte order* representation of binary datatypes was created as intermediate storage for binary data to be transmitted across the network. The idea is for each network program to

convert its own local binary data into network byte order before transmitting it. On the receiving side, the system must convert the incoming data from network byte order into its own internal byte order. This ensures that the binary data will be converted to the proper representation for the destination host. Figure 7.2 illustrates the process of network-byte-order conversion.

FIGURE 7.2: Using network byte order between hosts.



The .NET library includes methods to convert integer values to network byte order, and vice versa. These methods are included in the `IPAddress` class, contained in the `System.Net` namespace. One is `HostToNetworkOrder()`, which converts integer datatypes to a network byte order representation. In Listing 7.8, the `BinaryNetworkByteOrder.cs` program demonstrates using this method on integer datatypes.

Listing 7.8 The BinaryNetworkByteOrder.cs program

```
using System;
using System.Net;
using System.Text;

class BinaryNetworkByteOrder
{
   public static void Main()
   {
      short test1 = 45;
      int test2 = 314159;
      long test3 = -123456789033452;
      byte[] data = new byte[1024];
      string output;

      data = BitConverter.GetBytes(test1);
```

```
        output = BitConverter.ToString(data);
        Console.WriteLine("test1 = {0}, string = {1}", test1, output);

        data = BitConverter.GetBytes(test2);
        output = BitConverter.ToString(data);
        Console.WriteLine("test2 = {0}, string = {1}", test2, output);

        data = BitConverter.GetBytes(test3);
        output = BitConverter.ToString(data);
        Console.WriteLine("test3 = {0}, string = {1}", test3, output);

        short test1b = IPAddress.HostToNetworkOrder(test1);
        data = BitConverter.GetBytes(test1b);
        output = BitConverter.ToString(data);
        Console.WriteLine("test1 = {0}, nbo = {1}", test1b, output);

        int test2b = IPAddress.HostToNetworkOrder(test2);
        data = BitConverter.GetBytes(test2b);
        output = BitConverter.ToString(data);
        Console.WriteLine("test2 = {0}, nbo = {1}", test2b, output);

        long test3b = IPAddress.HostToNetworkOrder(test3);
        data = BitConverter.GetBytes(test3b);
        output = BitConverter.ToString(data);
        Console.WriteLine("test3 = {0}, nbo = {1}", test3b, output);
    }
}
```

The `BinaryNetworkByteOrder` program creates three types of integer data values and uses the `HostToNetworkOrder()` method to convert them to values in network byte order. The output from the `BinaryNetworkByteOrder` program on my machine is as follows:

```
C:\>BinaryNetworkByteOrder
test1 = 45, string = 2D-00
test2 = 314159, string = 2F-CB-04-00
test3 = -123456789033452, string = 14-CE-F1-79-B7-8F-FF-FF
test1 = 11520, nbo = 00-2D
test2 = 801833984, nbo = 00-04-CB-2F
test3 = 1499401231033958399, nbo = FF-FF-8F-B7-79-F1-CE-14

C:\>
```

You may notice something odd here. Notice that `HostToNetworkOrder()` returns the value in the same datatype as the original value. The byte values within the datatype are now placed in network byte order, ready for sending out on the network. Unfortunately, if the network byte order is not in the same binary representation as the local host, those data values will not be the same. For example, the value assigned to the `test1` variable is 45. When `test1` is converted to network byte order, it is assigned to the variable *`test1b`*. Now, the variable *`test1b`* is still a valid

short integer variable, but has the value 11520. This is obviously not the same as the original value of 45. When *test1b* is transmitted across the network, it must be converted back to the local host order to get the original value of 45.

WARNING Remember that when data is converted to network byte order, it may not have the same value as the original data value. The network byte order is only used for transporting the data across the network.

Before the destination host can use the data received, it must convert the data to the local binary datatype representation of the host.

Reading Data in Network Byte Order

After the integer values are converted to network byte order and sent to the remote system, they must be converted back to the host byte order representation so their original values can be used in the program. The `NetworkToHostOrder()` method of the `IPAddress` class converts data received in network byte order back to the appropriate byte order of the system running the program. Similar to `HostToNetworkOrder()`, the `NetworkToHostOrder()` method converts an integer value in network byte order to an integer value in the local host's byte order. It is possible that both orders are the same and no conversion will be necessary, but to be on the safe side, it is always best to include this method.

Sample Programs

Listing 7.9 is the `NetworkOrderClient.cs` program, which demonstrates how to use the `HostToNetworkOrder()` and `NetworkToHostOrder()` methods to transmit data across the network.

Listing 7.9 The NetworkOrderClient.cs program

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class NetworkOrderClient
{
   public static void Main()
   {
      byte[] data = new byte[1024];
      string stringData;
      TcpClient server;

      try
```

```
        {
           server = new TcpClient("127.0.0.1", 9050);
        } catch (SocketException)
        {
           Console.WriteLine("Unable to connect to server");
           return;
        }
        NetworkStream ns = server.GetStream();

        int recv = ns.Read(data, 0, data.Length);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        Console.WriteLine(stringData);

        short test1 = 45;
        int test2 = 314159;
        long test3 = -123456789033452;

        short test1b = IPAddress.HostToNetworkOrder(test1);
        data = BitConverter.GetBytes(test1b);
        Console.WriteLine("sending test1 = {0}", test1);
        ns.Write(data, 0, data.Length);
        ns.Flush();

        int test2b = IPAddress.HostToNetworkOrder(test2);
        data = BitConverter.GetBytes(test2b);
        Console.WriteLine("sending test2 = {0}", test2);
        ns.Write(data, 0, data.Length);
        ns.Flush();

        long test3b = IPAddress.HostToNetworkOrder(test3);
        data = BitConverter.GetBytes(test3b);
        Console.WriteLine("sending test3 = {0}", test3);
        ns.Write(data, 0, data.Length);
        ns.Flush();

        ns.Close();
        server.Close();
     }
  }
```

The `NetworkOrderClient` program uses the `TcpClient` class to create a TCP connection to a server. It then creates a `NetworkStream` object to send and receive data with the remote server. Once the connection is established, it sets values for three integer datatypes and sends them in network byte order to the server.

The `NetworkOrderSrvr.cs` program, shown in Listing 7.10, is used to receive the data and convert it back to host byte order.

Listing 7.10 The NetworkOrderSrvr.cs program

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class NetworkOrderSrvr
{
   public static void Main()
   {
      int recv;
      byte[] data = new byte[1024];

      TcpListener server = new TcpListener(9050);
      server.Start();
      Console.WriteLine("waiting for a client...");

      TcpClient client = server.AcceptTcpClient();
      NetworkStream ns = client.GetStream();

      string welcome = "Welcome to my test server";
      data = Encoding.ASCII.GetBytes(welcome);
      ns.Write(data, 0, data.Length);
      ns.Flush();

      data = new byte[2];
      recv = ns.Read(data, 0, data.Length);
      short test1t = BitConverter.ToInt16(data, 0);
      short test1 = IPAddress.NetworkToHostOrder(test1t);
      Console.WriteLine("received test1 = {0}", test1);

      data = new byte[4];
      recv = ns.Read(data, 0, data.Length);
      int test2t = BitConverter.ToInt32(data, 0);
      int test2 = IPAddress.NetworkToHostOrder(test2t);
      Console.WriteLine("received test2 = {0}", test2);

      data = new byte[8];
      recv = ns.Read(data, 0, data.Length);
      long test3t = BitConverter.ToInt64(data, 0);
      long test3 = IPAddress.NetworkToHostOrder(test3t);
      Console.WriteLine("received test3 = {0}", test3);

      ns.Close();
      client.Close();
      server.Stop();
   }
}
```

The NetworkOrderSrvr program uses the TcpListener class to listen on TCP port 9050 for incoming connection attempts. When a connection attempt is received, the program creates a TcpClient object. It then uses the GetStream() method to create a NetworkStream object for sending and receiving data from the remote host.

After the network connection is established and a welcome banner message is sent, the NetworkOrderSrvr expects to receive three integer datatypes from the remote host. Keep in mind that TCP is a stream-oriented communications channel and thus there is no guarantee that the three datatypes will be sent in three separate messages. To compensate for this, the NetworkOrderSrvr reads a set number of bytes from the NetworkStream for each datatype. This ensures that no matter how the data is received, it will be read from the TCP buffer in the right sizes.

Once the data is read from the TCP buffer, it is converted to the appropriate binary datatype using the BitConverter methods:

```
data = new byte[2];
recv = ns.Read(data, 0, data.Length);
short test1t = BitConverter.ToInt16(data, 0);
```

Remember, because you are not sure if the network byte order is correct for the system, the converted value should not be directly used; it is just temporary. To create the correct value, you must convert it to the host byte order:

```
short test1 = IPAddress.NetworkToHostOrder(test1t);
```

This ensures that the data value is in the correct binary representation for the host system on which the program is running.

The output from the NetworkOrderClient and NetworkOrderSrvr programs should be similar. The output from the client program should look like this:

```
C:\>NetworkOrderClient
Welcome to my test server
sending test1 = 45
sending test2 = 314159
sending test3 = -123456789033452

C:\>
```

And the output from the NetworkOrderSrvr program should be as follows:

```
C:\>NetworkOrderSrvr
waiting for a client...
received test1 = 45
received test2 = 314159
received test3 = -123456789033452

C:\>
```

The binary data values shown from the server program should be the same as those sent from the client program, showing that the data bytes were converted, sent, and reconverted to the proper order.

As usual, if you are running this program across the network, you can use the WinDump or Analyzer programs to watch the individual packets. By comparing the packet data bytes in network byte order from the `BinaryNetworkOrder` program against the Analyzer trace packets, you can see what data values are sent in which packets.

In my sample trace, the TCP data bytes show that the network byte order values `test2` (00 04 CB 2F) and `test3` (FF FF 8F B7 79 F1 CE 14) were sent in the same TCP packet. (Compare these values to those in the Listing 7.8 Binary Network ByteOrder output.) When these values are received, they are converted back to host byte order to retrieve the original values.

Moving Complex Objects

Now that you can send individual binary data values to a remote host, you may be wondering about the next step: sending groups of values across the network to a remote host. This section describes how to send groups of data as a single element to a remote device and how to decode the data back and retrieve the proper data values on the other end.

Creating a Collective Data Class

One common way to move groups of multiple data values between systems on a network is to create a class that contains all the data, along with a specific method for converting the data into a byte array. The basic class contains variables for the data elements used in the communication. For example:

```
class Employee
{
   public int EmployeeID;
   public string LastName;
   public string FirstName;
   public int YearsService;
   public double Salary;

   public int LastNameSize;
   public int FirstNameSize;
   public int size;
}
```

Here, the class `Employee` can be considered similar to a record, with the variables representing the fields in the record. Each instance of the class represents a record in the database.

Because the two string elements can have variable lengths, you should include additional elements to define the size of those elements. This is comparable to the variable text field methods shown in Chapter 5.

Eventually, a data element is created to hold the size of the total byte representation of the class instance—again a necessity because the class instance itself will be a variable length.

The GetBytes() Method

With the "collective" data class in place, you create a `GetBytes()` method for the class to help in converting all of the elements into a single byte array, suitable for sending out on the network. It looks like this:

```
public byte[] GetBytes()
{
   byte[] data = new byte[1024];
   int place = 0;
   Buffer.BlockCopy(BitConverter.GetBytes(EmployeeID), 0, data, place, 4);
   place += 4;
   Buffer.BlockCopy(BitConverter.GetBytes(LastName.Length), 0, data, place, 4);
   place += 4;
   Buffer.BlockCopy(Encoding.ASCII.GetBytes(LastName), 0,
       data, place, LastName.Length);
   place += LastName.Length;
   Buffer.BlockCopy(BitConverter.GetBytes(FirstName.Length),
       0, data, place, 4);
   place += 4;
   Buffer.BlockCopy(Encoding.ASCII.GetBytes(FirstName), 0,
       data, place, FirstName.Length);
   place += FirstName.Length;
   Buffer.BlockCopy(BitConverter.GetBytes(YearsService), 0, data, place, 4);
   place += 4;
   Buffer.BlockCopy(BitConverter.GetBytes(Salary), 0, data, place, 8);
   place += 8;
   size = place;
   return data;
}
```

The `GetBytes()` method performs three functions:

- It converts each element of the class to a byte array.
- It places all of the individual byte arrays into a single-byte array.
- It calculates the total size of the byte array.

By now you are familiar with the `BitConverter` methods to convert the various binary datatypes to byte arrays. What you may not be familiar with is the `Buffer` class's `BlockCopy()`

method. The `BlockCopy()` method allows you to copy an entire byte array into a location within another byte array. Here is the format of this method:

```
BlockCopy(byte[] array1, int start, byte[] array2, int offset, int size)
```

The *array1* parameter is the array to copy to *array2*. The starting location of the copy within *array1* is always the first byte. The `offset` within the second array changes after each item is added to the array. Each short integer value added takes up 2 bytes, and the double floating-point value takes up 8 bytes.

The unknowns are the two variable-length string values. This is where string-size elements from the class come in handy. Because you know how long the string instance is, you can use that value when placing it in the byte array, as illustrated in Figure 7.3.

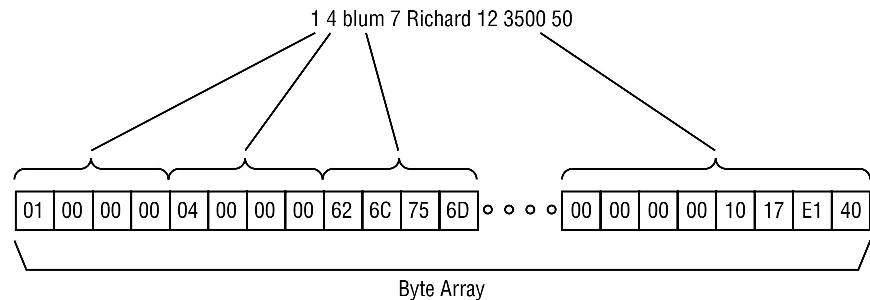


FIGURE 7.3: Placing data values within a byte array

The idea is to drop each byte array into its proper place in the data array. Care must be taken when calculating the location to ensure that each value is placed in the correct order in the byte array. Once the byte array is completed, it is ready to be sent across the network.

The Constructors

There should be two constructor formats for the data class. One is the default constructor, used to manually enter values into the data elements. The other reads a byte array produced from the `GetBytes()` method and converts it back into a class instance:

```
public Employee()
{
}

public Employee(byte[] data)
```

```
{
   int place = 0;
   EmployeeID = BitConverter.ToInt32(data, place);
   place += 4;
   LastNameSize = BitConverter.ToInt32(data, place);
   place += 4;
   LastName = Encoding.ASCII.GetString(data, place, LastNameSize);
   place = place + LastNameSize;
   FirstNameSize = BitConverter.ToInt32(data, place);
   place += 4;
   FirstName = Encoding.ASCII.GetString(data, place, FirstNameSize);
   place += FirstNameSize;
   YearsService = BitConverter.ToInt32(data, place);
   place += 4;
   Salary = BitConverter.ToDouble(data, place);
   }
```

The default constructor allows you to manually specify each of the data elements for a class instance, storing values in the instance. The second constructor format walks through the byte array and extracts each data element value. The variable-length string fields require the size parameters to help determine how many bytes are allocated for each string. Because the sizes were embedded into the byte array by the `GetBytes()` method, it is important to read each one and extract the proper number of bytes for the string.

The Whole Class Program

Putting all of the elements and methods together produces the `Employee` class file, `Employee.cs`, shown in Listing 7.11.

Listing 7.11 The Employee.cs program

```
using System;
using System.Text;

class Employee
{
   public int EmployeeID;
   private int LastNameSize;
   public string LastName;
   private int FirstNameSize;
   public string FirstName;
   public int YearsService;
   public double Salary;
   public int size;

   public Employee()
```

```
      {
      }

      public Employee(byte[] data)
      {

         int place = 0;
         EmployeeID = BitConverter.ToInt32(data, place);
         place += 4;
         LastNameSize = BitConverter.ToInt32(data, place);
         place += 4;
         LastName = Encoding.ASCII.GetString(data, place, LastNameSize);
         place = place + LastNameSize;
         FirstNameSize = BitConverter.ToInt32(data, place);
         place += 4;
         FirstName = Encoding.ASCII.GetString(data, place, FirstNameSize);
         place += FirstNameSize;
         YearsService = BitConverter.ToInt32(data, place);
         place += 4;
         Salary = BitConverter.ToDouble(data, place);

      }

      public byte[] GetBytes()
      {

         byte[] data = new byte[1024];
         int place = 0;
         Buffer.BlockCopy(BitConverter.GetBytes(EmployeeID), 0, data, place, 4);
         place += 4;
         Buffer.BlockCopy(BitConverter.GetBytes(
               LastName.Length), 0, data, place, 4);
         place += 4;
         Buffer.BlockCopy(Encoding.ASCII.GetBytes(
             LastName), 0, data, place, LastName.Length);
         place += LastName.Length;
         Buffer.BlockCopy(BitConverter.GetBytes(
               FirstName.Length), 0, data, place, 4);
         place += 4;
         Buffer.BlockCopy(Encoding.ASCII.GetBytes(
               FirstName), 0, data, place, FirstName.Length);
         place += FirstName.Length;
         Buffer.BlockCopy(BitConverter.GetBytes(YearsService), 0, data, place, 4);
         place += 4;
         Buffer.BlockCopy(BitConverter.GetBytes(Salary), 0, data, place, 8);
         place += 8;
         size = place;
         return data;
      }
   }
```

Because it is just a data container and can't run by itself, the `Employee.cs` program does not contain a `Main()` method. You can't compile the `Employee.cs` program by itself with the `csc` command. Instead, it must be compiled along with whatever programs use the `Employee` class.

Using Data Classes

Once the `Employee.cs` program is created, it's a snap to put it to work in client and server programs. Listing 7.12 shows a sample TCP client program that uses the `Employee` class to send employee information to the server.

Listing 7.12 The EmployeeClient.cs program

```
using System;
using System.Net;
using System.Net.Sockets;


class EmployeeClient
{
   public static void Main()
   {
      Employee emp1 = new Employee();
      Employee emp2 = new Employee();
      TcpClient client;

      emp1.EmployeeID = 1;
      emp1.LastName = "Blum";
      emp1.FirstName = "Katie Jane";
      emp1.YearsService = 12;
      emp1.Salary = 35000.50;

      emp2.EmployeeID = 2;
      emp2.LastName = "Blum";
      emp2.FirstName = "Jessica";
      emp2.YearsService = 9;
      emp2.Salary = 23700.30;

      try
      {
         client = new TcpClient("127.0.0.1", 9050);
      } catch (SocketException)
      {
         Console.WriteLine("Unable to connect to server");
         return;
      }
      NetworkStream ns = client.GetStream();

      byte[] data = emp1.GetBytes();
```

```
        int size = emp1.size;
        byte[] packsize = new byte[2];
        Console.WriteLine("packet size = {0}", size);
        packsize = BitConverter.GetBytes(size);
        ns.Write(packsize, 0, 2);
        ns.Write(data, 0, size);
        ns.Flush();

        data = emp2.GetBytes();
        size = emp2.size;
        packsize = new byte[2];
        Console.WriteLine("packet size = {0}", size);
        packsize = BitConverter.GetBytes(size);
        ns.Write(packsize, 0, 2);
        ns.Write(data, 0, size);
        ns.Flush();

        ns.Close();
        client.Close();
     }
  }
```

After two instances of the `Employee` class are created, data is entered into the data elements. The `GetByte()` method then converts the data into a byte array to send to the server. Before the byte array is sent, the size of the array is sent so the server knows how many bytes of data to read to complete the data package.

Similarly, the `EmployeeSrvr.cs` program, Listing 7.13, performs the server function using the `Employee` class.

Listing 7.13 The EmployeeSrvr.cs program

```
using System;
using System.Net;
using System.Net.Sockets;

class EmployeeSrvr
{
   public static void Main()
   {
      byte[] data = new byte[1024];
      TcpListener server = new TcpListener(9050);

      server.Start();
      TcpClient client = server.AcceptTcpClient();
      NetworkStream ns = client.GetStream();

      byte[] size = new byte[2];
      int recv = ns.Read(size, 0, 2);
      int packsize = BitConverter.ToInt16(size, 0);
```

```
        Console.WriteLine("packet size = {0}", packsize);
        recv = ns.Read(data, 0, packsize);

        Employee emp1 = new Employee(data);
        Console.WriteLine("emp1.EmployeeID = {0}", emp1.EmployeeID);
        Console.WriteLine("emp1.LastName = {0}", emp1.LastName);
        Console.WriteLine("emp1.FirstName = {0}", emp1.FirstName);
        Console.WriteLine("emp1.YearsService = {0}", emp1.YearsService);
        Console.WriteLine("emp1.Salary = {0}\n", emp1.Salary);

        size = new byte[2];
        recv = ns.Read(size, 0, 2);
        packsize = BitConverter.ToInt16(size, 0);
        data = new byte[packsize];
        Console.WriteLine("packet size = {0}", packsize);
        recv = ns.Read(data, 0, packsize);

        Employee emp2 = new Employee(data);
        Console.WriteLine("emp2.EmployeeID = {0}", emp2.EmployeeID);
        Console.WriteLine("emp2.LastName = {0}", emp2.LastName);
        Console.WriteLine("emp2.FirstName = {0}", emp2.FirstName);
        Console.WriteLine("emp2.YearsService = {0}", emp2.YearsService);
        Console.WriteLine("emp2.Salary = {0}", emp2.Salary);

        ns.Close();
        client.Close();
        server.Stop();

    }
}
```

The `EmployeeSrvr` program reads 2 bytes from the network, then converts them into an integer size value. The size value represents how many bytes to read for the data package. Once the data package is read, it can be converted to an `Employee` class instance using the `Employee` constructor.

To compile both the `EmployeeClient.cs` and `EmployeeSrvr.cs` programs, you must also include the `Employee.cs` file:

```
csc EmployeeClient.cs Employee.cs
csc EmployeeSrvr.cs Employee.cs
```

After compiling the two programs, you can test them out by starting `EmployeeSrvr` in a command-prompt window and `EmployeeClient` program in either a separate command-prompt window or on a separate network client. The output from the `EmployeeSrvr` program should look like this:

```
C:\>EmployeeSrvr
packet size = 30
```

```
emp1.EmployeeID = 1
emp1.LastName = Blum
emp1.FirstName = Katie Jane
emp1.YearsService = 12
emp1.Salary = 35000.5

packet size = 27
emp2.EmployeeID = 2
emp2.LastName = Blum
emp2.FirstName = Jessica
emp2.YearsService = 9
emp2.Salary = 23700.3

C:\>
```

The client program successfully transferred the employee data for each instance to the server program.

NOTE You may have noticed that the Employee examples did not use the network byte order to transfer the data values. These programs will only work on like machines on the network. You can experiment with converting the byte values to network byte order in order to make the examples run on any platform on the network.

NOTE Note: What we just wrote is a *serializer*—it serializes a class into a binary stream. .NET offers its own serializers, and we'll cover them in Chapter 16, "Using .NET Remoting."

Summary

In this chapter, you explored the .NET helper classes that are used in creating network programs. Whereas the `Socket` class allows you to manually create network programs using traditional Unix network programming methods, the three classes in this chapter—`TcpClient`, `TcpListener`, and `UdpClient`—help you produce network programs with a minimum amount of coding. The `TcpClient` and `TcpListener` classes are used for creating TCP network programs, and the `UDPClient` class for UDP programs.

You must be able to make the sent data intelligible to the receiving system. Text data is usually not a problem, but there are particular challenges to sending binary data. The C# language offers many types of binary data that must be converted to a byte array before sending to a remote system. The `BitConverter` class does this work. Once the data is converted to a byte array, it can be transmitted across the network to a remote system using one of the network classes.

Not all systems use the same method of representing binary data. The order in which bytes of multibyte values are stored is crucial to interpreting the data. Systems that use the big endian storage method cannot immediately interpret data from systems that use the little endian method. Communicating the binary data accurately requires conversion to a generic network byte order before sending the data. On the remote system, the converted data must be decoded from network byte order to the local system's byte order.

Transmitting complex data classes across the network can also be difficult. The simplest method is to convert each data element individually into a byte array and combine the byte arrays into a single large-byte array for transmission. After the array is received, it must be reassembled back into the original data class.