**CSCI 367 - Computer Networks I (5)**

TCP/IP Sockets – Echo Client and Server

Note: If using Visual Studio Code, client and server source code files must be in separate directories, and each directory must include the vscode directory. From the command-line, launch Visual Studio Code from each directory:

~/code/Server$ ls -a

. .. .vscode Program.c

~/code/Server$ code .

~/code/Client$ ls -a

. .. .vscode Program.c

~/code/Code$ code .

1. A socket is a programming abstraction used in client-server software applications. Sockets hide low-level network communication details, and makes it easier to write software applications that send and receive data over the network. Sockets are one of the main network communication components used to program network applications.

The following client program uses sockets to communicate with a daytime server:

Socket:

<https://man7.org/linux/man-pages/man2/socket.2.html>

Accept:

<https://www.geeksforgeeks.org/accept-system-call/>

<https://gcc.gnu.org/onlinedocs/gcc/Restricted-Pointers.html>

int accept(int \_\_fd, struct sockaddr \*\_\_restrict\_\_ \_\_addr, socklen\_t \*\_\_restrict\_\_ \_\_addr\_len)

Await a connection on socket FD.

When a connection arrives, open a new socket to communicate with it,

set \*ADDR (which is \*ADDR\_LEN bytes long) to the address of the connecting

peer and \*ADDR\_LEN to the address's actual length, and return the

new socket's descriptor, or -1 for errors.

This function is a cancellation point and therefore not marked with

\_\_THROW.

using System;

using System.Net;

using System.Net.Sockets;

using System.Text;

Server

#include <stdio.h>      //perror, printf

#include <stdlib.h>     // exit, atoi

#include <unistd.h>     // read, write, close

#include <arpa/inet.h>  // sockaddr\_in, AF\_INET, SOCK\_STREAM, INADDR\_ANY

#include <string.h>     // memset

#define TRUE 1

#define SERVER\_PORT 1234

#define RECEIVE\_BUFFER\_SIZE 1024

int main(int argc, char const \*argv[])

{

  struct sockaddr\_in server;

  struct sockaddr\_in client;

  char receive\_buffer[RECEIVE\_BUFFER\_SIZE];

  int serverFd = socket(AF\_INET, SOCK\_STREAM, 0);

  server.sin\_family = AF\_INET;

  server.sin\_addr.s\_addr = INADDR\_ANY;

  server.sin\_port = htons(SERVER\_PORT);

  int socket\_length = sizeof(server);

  int result = bind(serverFd, (struct sockaddr \*)&server, socket\_length);

  result = listen(serverFd, 10);

  while (TRUE)

  {

    socket\_length = sizeof(client);

    printf("Waiting for client connection...\n");

    int clientFd = accept(serverFd, (struct sockaddr \*)&client, &socket\_length);

    char \*client\_ip = inet\_ntoa(client.sin\_addr);

    printf("Accepted connection: %s:%d\n", client\_ip, ntohs(client.sin\_port));

    memset(receive\_buffer, 0, sizeof(receive\_buffer));

    int size = read(clientFd, receive\_buffer, sizeof(receive\_buffer));

    printf("Client message received: %s \n", receive\_buffer);

    result = write(clientFd, receive\_buffer, size);

    close(clientFd);

  }

  close(serverFd);

  return 0;

}

Client

#include <stdio.h>      // perror, printf

#include <stdlib.h>     // exit, atoi

#include <unistd.h>     // write, read, close

#include <arpa/inet.h>  // sockaddr\_in, AF\_INET, SOCK\_STREAM, INADDR\_ANY

#include <string.h>     // strlen, memset

#define SERVER\_IP\_ADDRESS "127.0.0.1"

#define SERVER\_PORT 1234

#define RECEIVE\_BUFFER\_SIZE 1024

const char message[] = "Hello!\n";

int main(int argc, char const \*argv[])

{

  int serverFd;

  struct sockaddr\_in server;

  char \*server\_ip\_address = SERVER\_IP\_ADDRESS;

  serverFd = socket(AF\_INET, SOCK\_STREAM, 0);

  server.sin\_family = AF\_INET;

  server.sin\_addr.s\_addr = inet\_addr(server\_ip\_address);

  server.sin\_port = htons(SERVER\_PORT);

  int socket\_length = sizeof(server);

  int result = connect(serverFd, (struct sockaddr \*)&server, socket\_length);

  int byte\_count = write(serverFd, message, strlen(message));

  char recieve\_buffer[RECEIVE\_BUFFER\_SIZE];

  memset(recieve\_buffer, 0, sizeof(recieve\_buffer));

  byte\_count = read(serverFd, recieve\_buffer, sizeof(recieve\_buffer));

  printf("Server message received: %s \n", recieve\_buffer);

  close(serverFd);

  return 0;

}

The program uses a socket to connect to a server program on a remote networked computer. It then sends a daytime request to the server, and receives a response. Following are two diagrams that illustrate the client-server communication cycle. First, the client sends a request for the current date and time:

216.456.1.41



129.6.15.28



4000

2025

6072

13

80

23

Client Computer

Server Computer

Daytime Server Port

Client Port for Daytime Server Data

Then the server returns a response that includes the current date and time:

216.456.1.41



129.6.15.28



4000

2025

6072

13

80

23

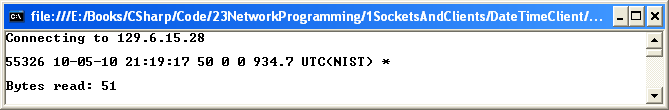
Client Computer

Server Computer

Daytime Server Port

Client Port for Daytime Server Data

When I run the program, it produces the following output:



The preceding output displays the current date and time, as well as the number of bytes sent by the server program. The server computer that sent this response is associated with the URL given by time-a.nist.gov. This URL maps to the IP address 129.6.15.28, that’s associated with a computer operated by the National Institute of Standards and Technology. According to their website:

The NIST Internet Time Service (ITS) allows users to synchronize computer clocks via the Internet. The time information provided by the service is directly traceable to UTC(NIST). The service responds to time requests from any Internet client in several formats including the DAYTIME, TIME, and NTP protocols.

Like all daytime server applications, this server application listens on port 13, the well-known port for a daytime server.

Let’s now analyze the program in more detail. First, to send a request to the daytime server, the client first needs to know where the server is located on the network. As mentioned above, a computer’s IP address serves this purpose. In the program above, the server computer’s address is given by the following dotted decimal value: 129.6.15.28. This IP address is associated with a computer on the internet that has an application that processes requests for the day and time. Using this IP address, routers on the internet route the request to the destination computer. Once the destination computer receives the request, the port number that’s encapsulated in the TCP part of the network packet is extracted by the server’s TCP/IP system software. The port number allows the TCP/IP software to direct the TCP segment’s data portion to the appropriate server program.

The IP address and port number are assigned to constants:

const string IP\_ADDRESS = "129.6.15.28";

const int PORT = 13;

Note that the constant IP\_ADDRESS is assigned a string value and the constant PORT is assigned an integer value.

Next, a message is output to the monitor telling us the status of the program:

Console.WriteLine("Connecting to " + IP\_ADDRESS);

Then, an IPAddress object reference is returned from the call to IPAddress.Parse:

IPAddress address = IPAddress.Parse(IP\_ADDRESS);

The IPAddress class is in the System.Net namespace. An IPAddress object represents the IP address of the remote host. The Parse method receives a string that represents a dotted decimal IP address value, as in 129.6.15.28, and returns a reference to an IPAddress object.

Next, an IPEndPoint object is allocated:

EndPoint endPoint = new IPEndPoint(address, PORT);

The IPEndPoint class is in the System.Net namespace. The IPEndPoint constructor takes two parameters: a reference to an IPAddress object, and an integer value that represents the port number of the server program. An IPEndPoint object encapsulates the destination computer’s IP address and the port number of the server program.

Once we have the IP address of the computer and the port number of the service, we then allocate a socket object:

Socket client = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

The Socket class is in the System.Net.Sockets namespace. The first constructor parameter, AddressFamily.InterNetwork, is an enumeration value that designates the addressing scheme used by the socket. The enumeration value AddressFamily.InterNetwork maps to IP version 4 addresses. There are many addressing schemes, but the most common are IP version 4 (IP4) and IP version 6 (IP6). IP4 uses four bytes to represent an IP address, and IP6 uses six bytes.

Obviously, many more IP addresses can be represented using six bytes rather than four bytes. As a matter of fact, the IP6 addressing scheme was developed because of the growth in the number of devices connected to the internet. This increase in connected devices creates a need for more IP addresses.

The second constructor parameter, SocketType.Stream, is an enumeration value that designates the type of socket. According to the .NET Class Library documentation, the enumeration value SocketType.Stream:

Supports reliable, two-way, connection-based byte streams without the duplication of data and without preservation of boundaries. A Socket of this type communicates with a single peer and requires a remote host connection before communication can begin. Stream uses the Transmission Control Protocol (Tcp) ProtocolType and the InterNetworkAddressFamily.

As noted in the documentation above, the SocketType.Stream enumeration value uses TCP as the transport layer protocol.

The third constructor parameter, ProtocolType.Tcp, is an enumeration value that designates the communication protocol used by the socket. In this case, the TCP protocol is specified. According to the .NET Class Library documentation:

The Socket class uses the ProtocolType enumeration to inform the Windows Sockets API of the requested protocol. Low-level driver software for the requested protocol must be present on the computer for the Socket to be created successfully.

Because the SocketType.Stream enumeration value defaults to the TCP protocol, it may seem that the SocketType.Stream and the ProtocolType.Tcp enumeration values are redundant. In this case they are, but this isn’t always the case. There are other SocketType enumeration values that can be used with a variety of ProtocolType enumeration values.

After a socket object is allocated, a connection is established with the server:

client.Connect(endPoint);

The Connect method takes a reference to an IPEndPoint object.

After the Socket’s Connect method returns, a reliable, byte-based TCP/IP communication channel is established. The client can then send data to the server and receive data from the server. In this case, the client wants to receive data from the server that represents the current date and time. The following statements accomplish this task:

byte[] buffer = new byte[BUFFER\_SIZE];

int bytesRead = client.Receive(buffer);

The first statement allocates an array of bytes, and assigns the array reference to the variable named buffer. BUFFER\_SIZE is a constant assigned the value 100:

const int BUFFER\_SIZE = 100;

This byte array is used to receive the data sent from the server. The reference variable buffer is passed to the Socket class’ Receive method. The Receive method fills the buffer with data sent back from the server, and returns a value that represents the number of bytes that were inserted in the buffer.

Next, the Socket’s Close method is executed:

client.Close();

The Close method terminates the connection with the server program.

Finally, the program uses the following statements to output the data to the client’s monitor:

string data = Encoding.ASCII.GetString(buffer, 0, bytesRead);

Console.WriteLine(data);

Console.WriteLine("Bytes read: " + bytesRead);

The Encoding class is in the System.Text namespace. The Encoding class’ ASCII property returns a reference to an Encoding object. The Encoding class’ GetString method returns a reference to a String object that represents the current date and time. The first parameter to GetString represents the byte array that contains the data sent from the server, the second parameter represents the starting index of the byte array that the method should start reading from, and the third parameter represents the number of byte values the method should read.

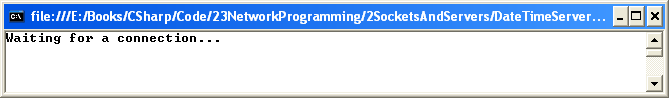
Note that in the preceding discussion, no mention was made of the client IP address, and the port used by the client application. If you look at the program’s source code, you won’t see any references to the client IP address or port number. This is because the TCP/IP system software on the client computer takes care of this for us. In other words, when the client TCP/IP system sends a network packet, it automatically encapsulates the client computer’s IP address, and assigns a temporary port number to the client program. In the preceding diagram that illustrates the request and response cycle, although I used a port number of 4000 for the client program, that probably wasn’t the actual port that was assigned to the program by the TCP/IP system software.

**23.3 Sockets and Servers**

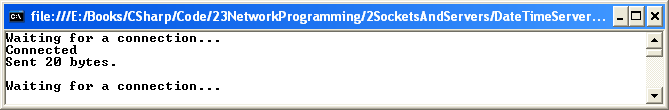
This next example demonstrates how to create our own server program. Here’s the server program:

Before I analyze the code, let’s study the output that the client and server produce.

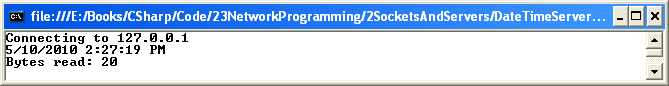
First, our server waits for a connection from the client:



When a connection is established, the server returns the current date and time, and then shuts down:



Here’s the output that the client produces:



Note that the amount of data sent is the same as the amount received: 20 bytes.

The following diagram illustrates the request and response cycles (I’ll discuss the significance of the IP address value 127.0.01 shortly).

First, the client makes a connection:

127.0.0.1



127.0.01



2025

5040

Client Computer

Server Computer

Daytime Server Port

Client Port for Daytime Server Data

Then the server returns the current date and time:

127.0.0.1



127.0.01



2025

5040

Client Computer

Server Computer

Daytime Server Port

Client Port for Daytime Server Data

Let’s now analyze the server. The port number used for our server is shown here:

const int PORT = 5040;

Note that since port values above 5000 are rarely reserved for special services, I chose port number 5040.

Next, I allocate IPAddress, EndPoint and Socket objects:

IPAddress address = IPAddress.Parse(IPAddress.Any);

EndPoint endPoint = new IPEndPoint(address, PORT);

Socket server = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

Note the value passed to the IPAddress.Parse method is IPAddress.Any. This value allows any client to connect to the server.

Next, I bind the EndPoint object to the Socket object using Socket’s Bind method, and call the Listen method:

server.Bind(endPoint);

server.Listen(5);

The server is now configured to listen for requests on port 5040. The parameter to Listen specifies the number of client connections that can be queued up at any one time. The TCP/IP system software handles the client queue.

Next, the scope of the while-loop is entered, and the server socket waits for a connection from a client:

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

Socket connection = server.Accept();

Console.WriteLine("Connected");

When a client sends a TCP message to initiate contact, the Socket’s Accept method returns a reference to a newly allocated Socket object. This Socket object encapsulates information about the client computer’s IP address and port number, so that the server can send back a response.

The server program then obtains the current data and time, populates a byte array, and sends the bytes to the client program using the Socket’s Send method:

DateTime dateTime = DateTime.Now;

string data = dateTime.ToString();

byte[] buffer = Encoding.ASCII.GetBytes(data);

int sentBytes = connection.Send(buffer);

The DateTime class in the System namespace comes in handy here. DateTime class’ Now property returns a reference to a DateTime object that encapsulates the current date and time. The DateTime class’ ToString method returns a reference to a String object that represents the date and time as a string of characters. The Encoding class’ ASCII property returns a reference to an Encoding object. The Encoding class’ GetBytes method receives a reference to a String object, and returns a reference to an array of bytes that represents the characters encapsulated in the String object.

The Send method receives the byte array reference, sends the bytes to the client, and returns the number of bytes sent. The number of bytes sent by the server program should be the same as the number of bytes received by the client program. As seen in the output above, this is the case.

After the data is sent by the server, the server closes the connection with the client, and outputs the number of bytes sent:

connection.Close();

Console.WriteLine("Sent " + sentBytes + " bytes." + Environment.NewLine);

Program control then transfers to the first statement in the while-loop, and the server waits for the next client connection. Note that the while-loop is defined to never terminate. This behavior is typical with server applications.

I didn’t have to make any modifications to the client program except change the IP\_ADDRESS and PORT values to 127.0.0.1 and 5040, as shown here:

const string IP\_ADDRESS = "127.0.0.1"; //Localhost

const int PORT = 5040;

The IP address 127.0.0.1 is called the loop-back address. Because I’m testing the server on the same machine as the client, I can use the loop-back address which is also referred to as the Localhost address. This IP address is reserved to make it easier to test client-server programs without having to consider the actual IP address of the computer the client and server are hosted on.

**23.4 TcpClient and NetworkStream Classes**

To make it easier to send and receive data over the network, the .NET Class Library includes the TcpClient and NetworkStream classes. The TcpClient class uses class containment (the has-a relationship) to encapsulate a reference to a Socket object. The TcpClient is designed to abstract away much of the details involved when sending or receiving data over a network connection. The following program illustrates the TcpClient and NetworkStream classes:

using System;

using System.Net.Sockets;

using System.Text;

class Program

{

static void Main()

{

const int BUFFER\_SIZE = 100;

const string IP\_ADDRESS = "129.6.15.28";

const int PORT = 13;

Console.WriteLine("Connecting to " + IP\_ADDRESS);

TcpClient client = new TcpClient(IP\_ADDRESS, PORT);

NetworkStream networkStream = client.GetStream();

byte[] buffer = new byte[BUFFER\_SIZE];

int bytesRead = networkStream.Read(buffer, 0, BUFFER\_SIZE);

networkStream.Close();

string data = Encoding.ASCII.GetString(buffer, 0, bytesRead);

Console.WriteLine(data);

Console.WriteLine("Bytes read: " + bytesRead);

Console.Read();

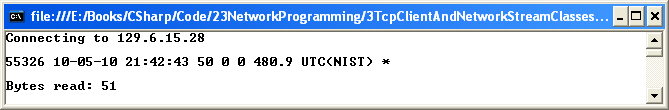
}

}

Code Listing 23.3

The TcpClient and NetworkStream classes are part of the System.Net.Sockets namespace.

The program produces the following output:



This client program is essentially the same as the client program that used the Socket class. The only difference is that I used the TcpClient and NetworkStream classes to simplify the program.

Let’s analyze the program. First, a TcpClient object is allocated. The constructor is passed the IP address of the server computer and the port number of the server application:

TcpClient client = new TcpClient(IP\_ADDRESS, PORT);

Then, the TcpClient.GetStream method is called:

NetworkStream networkStream = client.GetStream();

This method uses the TcpClient class’ encapsulated Socket object to connect with the server. The GetStream method returns a reference to a NetworkStream object, which is then used to read the data sent by the server:

byte[] buffer = new byte[BUFFER\_SIZE];

int bytesRead = networkStream.Read(buffer, 0, BUFFER\_SIZE);

The Read method is passed a reference to a byte array, the starting offset in the array where the data is to be written, and the size of the array. The Read method returns the number of bytes read, as can be seen in the output above.

Next, the NetworkStream is closed:

networkStream.Close();

These Close methods release any system resources associated with the NetworkStream and TcpClient objects. Note that we don’t have to call the TcpClient’s Close method. According to the .NET Class Library documentation, calling the NetworkStream’s Close method:

Closes the current stream and releases any resources (such as sockets and file handles) associated with the current stream.

Next, after the Read method reads the data and populates the byte array, the byte array is converted to a String object using the Encoding class’ GetString method that we discussed earlier:

string data = Encoding.ASCII.GetString(buffer, 0, bytesRead);

Finally, the data is output:

Console.WriteLine(data);

Console.WriteLine("Bytes read: " + bytesRead);

**23.6 Echo Server and Client**

Echo servers are a common way to test a client-server application. The client sends a message to the server and the server echoes the message back. This process continues until the client terminates the session. As seen in the server program below, when the client sends the string exit to the server, the client connection is terminated, and the server waits for the next client connection. Here is the server program:

The main difference between echo server and the date-time server program we analyzed earlier is the inclusion of the following nested loop:

while (true)

{

byte[] buffer = new byte[BUFFER\_SIZE];

int bytesRead = networkStream.Read(buffer, 0, BUFFER\_SIZE);

if (bytesRead == 0) break;

string data = Encoding.ASCII.GetString(buffer, 0, bytesRead);

Console.WriteLine(data);

networkStream.Write(buffer, 0, bytesRead);

}

This loop is needed, because I made the buffer very small, just 20 bytes. If the number of bytes sent to the server is greater than 20 bytes, multiple calls to the NetworkStream’s Read method are needed before all the data is read. The Read method returns 0 when all the data is read.

I purposely made the buffer small to illustrate a point. Real-world network buffers are much larger.

Below is the client program:

using System;

using System.Net;

using System.Net.Sockets;

using System.Text;

class Program

{

static void Main()

{

const int BUFFER\_SIZE = 100;

const string IP\_ADDRESS = "127.0.0.1"; //Localhost

const int PORT = 5040;

TcpClient client = new TcpClient(IP\_ADDRESS, PORT);

NetworkStream networkStream = client.GetStream();

while (true)

{

Console.Write("Enter message or ‘exit’ to exit: ");

string message = Console.ReadLine();

if (message.Equals("exit")) break;

byte[] buffer = Encoding.ASCII.GetBytes(message);

networkStream.Write(buffer, 0, buffer.Length);

buffer = new byte[BUFFER\_SIZE];

int bytesRead = networkStream.Read(buffer, 0, BUFFER\_SIZE);

string data = Encoding.ASCII.GetString(buffer, 0, bytesRead);

Console.WriteLine(data);

}

networkStream.Close();

Console.Read();

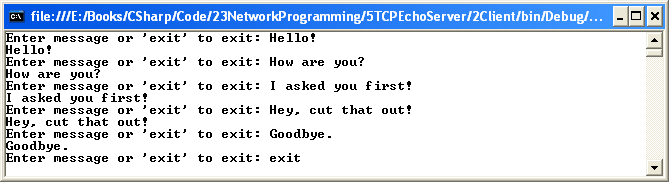
}

}

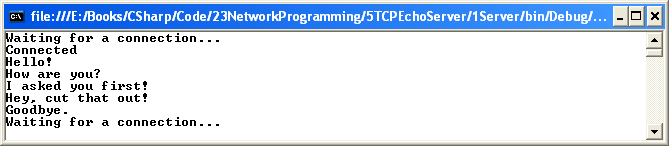
Code Listing 23.6

Inside the loop, the program prompts the user for a message and sends the message to the server. It then waits for a response from the server. When the server echoes the message back, the client outputs the echoed message to the monitor and re-prompts the user for the next message to send to the server. This continues until the user types in the string exit.

A sample run is shown below. Here’s the client’s output:



Here’s the server’s output:



**23.7 Multithreaded Server**

TCP/IP based server applications are designed to handle multiple client requests concurrently, and process these requests in a timely manner. The server application we’ll study next is designed to handle multiple client requests concurrently. The following code listing implements a multithreaded the server:

using System;

using System.Net;

using System.Net.Sockets;

using System.Text;

class Program

{

static void Main()

{

const int PORT = 5040;

EndPoint endPoint = new IPEndPoint(IPAddress.Any, PORT);

Socket server = new Socket(AddressFamily.InterNetwork, SocketType.Stream,

ProtocolType.Tcp);

server.Bind(endPoint);

server.Listen(5);

while (true)

{

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

Socket connection = server.Accept();

Console.WriteLine("Connected\n");

ClientConnection clientConnection = new ClientConnection(connection);

}

}

}

Code Listing 23.7

In both the client and server programs, rather than use the TcpClient and TcpListener classes, I decided to simply use the Socket class. This gives us practice using just the Socket class for network communication.

The Main method first allocates a stream-based TCP Socket object, binds the socket to an endpoint, and waits for a request. When a request is received, the Socket’s Accept method returns a reference to a Socket object that is then used to communicate with the client. This Socket reference is passed to the ClientConnection class’ constructor:

ClientConnection clientConnection = new ClientConnection(connection);

Let’s study the ClientConnection class in detail. The following code listing implements the ClientConnection class:

using System;

using System.Net;

using System.Net.Sockets;

using System.Text;

using System.Threading;

public class ClientConnection

{

private Socket connection;

public ClientConnection(Socket connection)

{

this.connection = connection;

Thread clientThread = new Thread(new ThreadStart(ProcessConnection));

clientThread.Start();

}

public void ProcessConnection()

{

DateTime dateTime = DateTime.Now;

string data = dateTime.ToString();

byte[] buffer = Encoding.ASCII.GetBytes(data);

int sentBytes = connection.Send(buffer);

IPEndPoint remoteEndPoint = (IPEndPoint)connection.RemoteEndPoint;

connection.Close();

Console.WriteLine("Sent " + sentBytes + " bytes to:");

Console.WriteLine("Remote IP Address: " + remoteEndPoint.Address);

Console.WriteLine("Remote Port: " + remoteEndPoint.Port + "\n");

}

}

Code Listing 23.8

The ClientConnection class’ constructor assigns the Socket reference to a field named connection that’s of type Socket:

this.connection = connection;

Then, to handle the request, a Thread object is allocated and started:

Thread clientThread = new Thread(new ThreadStart(ProcessConnection));

clientThread.Start();

A ThreadStart delegate reference is passed to the Thread constructor. The ProcessConnection method serves as the callback method.

So that the ProcessConnection method can access the Socket non-static field, the ProcessConnection method must be declared non-static.

The ProcessConnection method simply sends a response representing the date and time back to the client:

DateTime dateTime = DateTime.Now;

string data = dateTime.ToString();

byte[] buffer = Encoding.ASCII.GetBytes(data);

int sentBytes = connection.Send(buffer);

Then, so that we can see which client’s request has been processed, I output the client’s IP address and port number:

IPEndPoint remoteEndPoint = (IPEndPoint)connection.RemoteEndPoint;

connection.Close();

Console.WriteLine("Sent " + sentBytes + " bytes to:");

Console.WriteLine("Remote IP Address: " + remoteEndPoint.Address);

Console.WriteLine("Remote Port: " + remoteEndPoint.Port + "\n");

Below is the client application that’s used to test the server:

using System;

class Program

{

static void Main()

{

for (int clientCount = 0; clientCount < 5; clientCount++)

{

ServerConnection serverConnection = new ServerConnection();

}

Console.Read();

}

}

Code Listing 23.9

The Main method simply employs a loop to create five connections to the server. The ServerConnection class is where all the work is done. It’s shown here:

using System;

using System.Net;

using System.Net.Sockets;

using System.Text;

using System.Threading;

public class ServerConnection

{

private const int BUFFER\_SIZE = 100;

private const string IP\_ADDRESS = "127.0.0.1"; //Localhost

private const int PORT = 5040;

private static IPAddress address = IPAddress.Parse(IP\_ADDRESS);

private static EndPoint endPoint = new IPEndPoint(address, PORT);

private Socket connection = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

public ServerConnection()

{

Thread clientThread = new Thread(new ThreadStart(SendRequest));

clientThread.Start();

}

public void SendRequest()

{

connection.Connect(endPoint);

byte[] buffer = new byte[BUFFER\_SIZE];

int bytesRead = connection.Receive(buffer);

connection.Close();

string data = Encoding.ASCII.GetString(buffer, 0, bytesRead);

Console.WriteLine("Bytes read: " + bytesRead);

Console.WriteLine(data + "\n");

}

}

Code Listing 23.10

Each ServerConnection object has a field of type Socket that represents the connection to the server. The Socket field is initialized when the ServerConnection object is allocated:

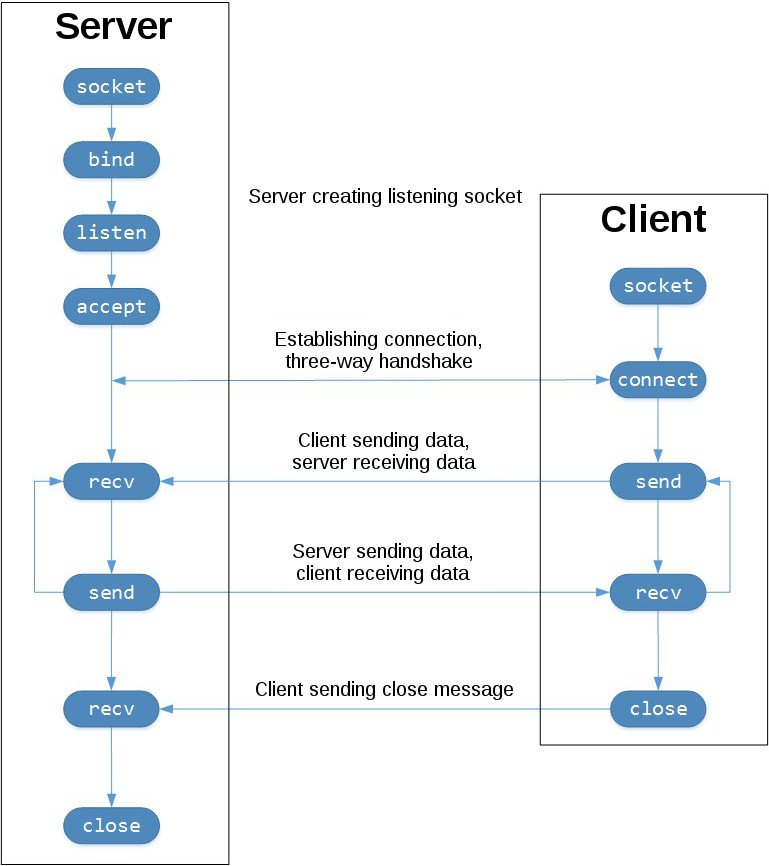
private Socket connection = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

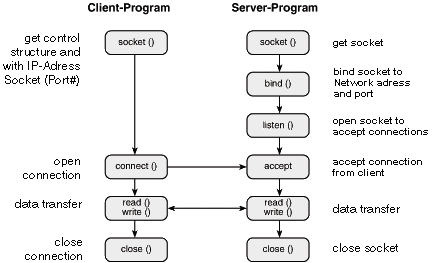
Next, the constructor allocates a Thread object that is used to send the request to the server. By allocating a separate thread for each request, multiple requests can be sent to the server concurrently.

A ThreadStart delegate is passed in to the Thread constructor, with the SendRequest method serving as the callback method. The SendRequest method connects to the server, and waits for a response that will consist of the current data and time.

Like any multithreaded application, when implementing a multithreaded server, you need to remember to take into account synchronization issues. As we learned when studying multithreaded programming, this can be tricky.

**Socket API: Client and Server Method-Call Sequence**





Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

**TCP/IP Layers: User Space vs Kernel Space vs NIC Device Driver**

Diagram

Description automatically generated