Multi-client Server Programming

Blocking server programs, although they allow connections of multiple clients, are meant to work with only one client at a time. Consider the following example of blocking server code:

```
serversock.Bind(ipep);  //ipep is the server end point
serversock.Listen(10);
Socket client = serversock.Accept();
```

After execution of the three lines of code, the server will wait for a client to be connected. Once the connection is established, other clients cannot access the server until this client socket is closed, i.e., clients can only be connected one at a time. This becomes a major problem when one of the clients is using most of the server time.

More desirable server would be a server that allows simultaneous connections of many clients. Two approaches can be used, which are

1. Poll()/Select() method

2. Multithreaded server

1. Poll()/Select() method

The .net Sockets library includes non-blocking socket methods found in the Unix socket library. This allows programmers to easily port Unix network programs to the Windows environment. The Poll()/Select() methods help network applications avoid getting stuck.

The Poll() Method

The format of the Poll() method is:

```
bool Poll(int microseconds, SelectMode mode);
```

It returns a true if the action would complete, or false if the action would block. The microseconds indicate the amount of time the Poll() method will wait and watch the socket for the indicated events. The SelectMode specifies the type of action to watch for.

The SelectMode.SelectRead value will cause the Poll() to return a true under the following conditions:

- If an Accept() method call would succeed
- If data is available on the socket
- If the connection has been closed

The SelectMode.SelectWrite value will cause the Poll() to return a turn under the following conditions:

- If a Connect() method call has succeeded
- If data can be sent on the socket

The SelectMode.SelectError value will cause the Poll() to return a true under the following conditions:

- If a Connect() method call has failed
- If out-of-band data is available and the Socket OutOfBandInline property has not been set

A sample server program based on the Poll() method

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

namespace ConsoleApplication1
{
    class tcpPollServer
    {
        static void Main(string[] args)
        {
            int recv;
            byte[] data = new byte[1023];
            IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);

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

            newsock.Bind(ipep);
            newsock.Listen(10);
            Console.WriteLine("Waiting for a client ...");
            bool result;
            int i = 0;
            while (true)
            {
                i++;
                Console.WriteLine("Polling for accept#{0}| ...", i);
                result = newsock.Poll(500000, SelectMode.SelectRead); //0.5 second
                if (result) break;
            }
            Socket client = newsock.Accept();

            IPEndPoint newclient = (IPEndPoint)client.RemoteEndPoint;
            Console.WriteLine("Connected with {0} at port {1}", newclient.Address,
newclient.Port);

            string welcome = "Welcome to Poll server";
            data = Encoding.ASCII.GetBytes(welcome);
            client.Send(data, data.Length, SocketFlags.None);

            i = 0;
            while (true)
            {
```

```
                Console.WriteLine("Polling for receive #{0}...", i);
                i++;
                result = client.Poll(3000000, SelectMode.SelectRead); //3 sec
                if (result)
                {
                    data = new byte[1024];
                    i = 0;
                    recv = client.Receive(data);
                    if (recv == 0) break;

                    Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
                    client.Send(data, recv, 0);
                }
            }

            Console.WriteLine("Disconnected from {0}", newclient.Address);
            client.Close();
            newsock.Close();

        }
    }
}
```

The Select() Method

The select() method polls one or more of sockets for blocking functions. As sockets become available for reading or writing, the Select() method can determine which ones are ready to use and which ones are would block if used. The format of this method is:

```
      Socket.Select(IList checkRead, IList checkWrite, IList checkError, int
microseconds)
```

The IList object represents a collection of objects than can be individually accessed by an index value.

A sample server program based on the Select() method

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

namespace ConsoleApplication1
{
    class tcpSelectServer
    {
        static void Main(string[] args)
        {
            ArrayList sockList = new ArrayList(2);
            ArrayList copyList = new ArrayList(2);
```

```
            Socket main = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
            IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9050);
            byte[] data = new byte[1024];
            string stringData;
            int recv;

            main.Bind(iep);
            main.Listen(2);

            Console.WriteLine("Waiting for 2 clients...");
            Socket client1 = main.Accept();
            IPEndPoint iep1 = (IPEndPoint)client1.RemoteEndPoint;
            client1.Send(Encoding.ASCII.GetBytes("Selcome to Select server"));
            Console.WriteLine("Connected to {0}", iep1.ToString());
            sockList.Add(client1);

            Console.WriteLine("Waiting for 1 more clients...");
            Socket client2 = main.Accept();
            IPEndPoint iep2 = (IPEndPoint)client2.RemoteEndPoint;
            client2.Send(Encoding.ASCII.GetBytes("Selcome to Select server"));
            Console.WriteLine("Connected to {0}", iep2.ToString());
            sockList.Add(client2);
            main.Close();

            while (true)
            {
                copyList = new ArrayList(sockList);
                Console.WriteLine("Monitoring {0} sockets...", copyList.Count);
                Socket.Select(copyList, null, null, 1000000);

                foreach(Socket client in copyList)
                {
                    data = new byte[1024];
                    recv = client.Receive(data);
                    stringData = Encoding.ASCII.GetString(data, 0, recv);
                    Console.WriteLine("Received: {0}", stringData);
                    if (recv == 0)
                    {
                        iep = (IPEndPoint)client.RemoteEndPoint;
                        Console.WriteLine("Client {0} disconnected.", iep.ToString());
                        client.Close();
                        sockList.Remove(client);
                        if (sockList.Count == 0)
                        {
                            Console.WriteLine("Last client disconnected, bye");
                            return;
                        }
                    }
                    else
                        client.Send(data, recv, SocketFlags.None);
                }
            }
        }
    }
}
```

If the sever socket receive actual data, it is echoed back to the originating client using the standard Send() method. After the for loop completes, the whole process can start over again.

2. Multithreaded Server

A drawback of using the Select() or Poll() method to handle multiple clients is that the code can become convoluted. Trying to accommodate multiple clients as they send and receive data at different intervals is quite complex in programming logic sense. Also, these approaches are in effect time shared blocking methods, and the clients must wait their turns. It would be nice if each client has a dedicated channel to talk to the sever at all times. Multithread facilitates this need.

The key to the multithreaded server is to create the main server Socket object in the main program. As each client connects to the server, the main program creates a separate thread to handle the connection.

A sample multi-threaded server program

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

namespace ConsoleApplication1
{
    class threadedServer
    {
        private TcpListener client;

        public threadedServer()
        {
            client = new TcpListener(IPAddress.Any, 9050);
            client.Start();

            Console.WriteLine("Waiting for clients...");
            while(true)
            {
                while(!client.Pending())
                {
                    Thread.Sleep(1000);
                }

                ConnectionThread newconnection = new ConnectionThread();
                newconnection.threadListner = this.client;
                Thread newthread = new Thread( new
ThreadStart(newconnection.HandleConnection));
                newthread.Start();

            }
        }
```

```
        public static void Main()
        {
            threadedServer server = new threadedServer();
        }
    }


    class ConnectionThread
    {
        public TcpListener threadListner;
        private static int connections =0;

        public void HandleConnection()
        {
            int recv;
            byte[] data = new byte[1024];
            TcpClient client = threadListner.AcceptTcpClient();
            NetworkStream ns = client.GetStream();
            connections++;
            Console.WriteLine("New client accepted: {0} active connections",
connections);

            string welcome = "Welcome to multithreaded 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;

                ns.Write(data, 0, recv);
            }
            ns.Close();
            client.Close();
            Console.WriteLine("Client disconnected: {0} active connections",
connections);

        }
    }
}
```

For additional topics on thread, please study the ThreadPool class.

Reference for this material: “C# Network Programming,” by Richard Blum, Sybex.