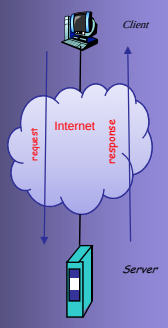


INTERNET PROTOCOLS AND CLIENT-SERVER PROGRAMMING SWE344

Fall Semester 2008-2009 (081)

Module 7: Asynchronous Programming & Multi-client Servers

Dr. El-Sayed El-Alfy
Computer Science Department
King Fahd University of Petroleum and Minerals
alfy@kfupm.edu.sa

Objectives

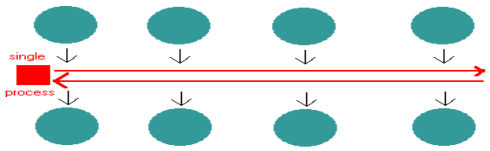
- Part 1:
  - Understand **asynchronous** methods of the **Socket** & **Stream** classes
  - Create **Asynchronous** client-server applications
  - Write User-Defined Asynchronous Methods
- Part 2:
  - Create multi-client TCP servers using **Thread** class
  - Create multi-client TCP servers using **ThreadPool** class
  - Create multi-client TCP servers using **asynchronous methods**

Asynchronous Methods

- Recall that the **Accept**, **Connect**, **Send** and **Receive** methods of the **Socket** class are blocking methods
- Similarly, the **Read** and **Write** methods of the **FileStream** and **NetworkStream** classes are blocking methods
- Pros: simple
- Cons: each of these methods blocks the execution of a program until its operation is completed.
  - may be acceptable in a console application, but undesirable in windows applications where controls must remain responsive while network communications are taking place
- Also recall that we have solved the problem of blocking by using threads.
  - Although it is easy, it can get very complicated very fast with worries about controlling access to shared resources

Asynchronous Methods …

- A more efficient way of avoiding blocking is by using the asynchronous methods of the **Socket** and **Stream** classes.



The concept of event-driven programming

- Asynchronous methods are always a pair: **BeginZZ** and **EndZZ** where ZZ is the corresponding synch. method
  - E.g., the asynchronous BeginAccept and EndAccept methods corresponds to the synchronous Accept method
- After calling BeginZZ, an application can continue executing instructions on the calling thread while the asynchronous operation takes place on a different thread
- For each call to BeginZZ, the application should also call EndZZ to get the results of the operation.

Asynchronous Methods …

- The Begin method takes two additional arguments to the arguments of the corresponding blocking method
  - An instance of **AsyncCallback** delegate
  - An instance of some class (type **Object),** used to pass state information
- When the task is finished, the Begin method invokes the delegate passed to it as argument to notify the caller.
- The delegate in turn calls its registered method, which will then call the **End** method to complete the task.
- The Begin method returns an object of type **IAsyncResult**
  - has **IsCompleted** property that can be used to monitor the progress of the asynchronous operation
  - has **AsyncState** property that can be used to recover the state information object passed to the Begin method

Asynchronous Methods …

- The signature of the **AsyncCallback** delegate is:

```
void AsyncCallback (IAsyncResult result)
```

  - Thus, to call the Begin method, we must first define a method with the above signature
  - Inside this method, you must call the End method to obtain the result of the asynchronous operation.
- The return type of the End is always the return type of the corresponding blocking method

Asynchronous Socket & Stream Methods

- **Asynchronous** methods of the **Socket** class:

IAsyncResult **BeginAccept** (AsyncCallback callback, object state) Socket **EndAccept** (IAsyncResult asyncResult)
IAsyncResult **BeginConnect** (EndPoint remoteEP, AsyncCallback callback, object state) void **EndConnect** (IAsyncResult asyncResult)
IAsyncResult **BeginReceive**(**byte**[] buffer, **int** offset, **int** size, SocketFlags socketFlags, AsyncCallback callback,object state) **int** **EndReceive**(IAsyncResult asyncResult)
IAsyncResult **BeginSend**(**byte**[] buffer,**int** offset,**int** size, SocketFlags socketFlags, AsyncCallback callback, object state) **int** **EndSend**(IAsyncResult asyncResult)

Note: BeginSendTo, EndSendTo, BeginReceiveFrom, EndReceiveFrom exist in similar format

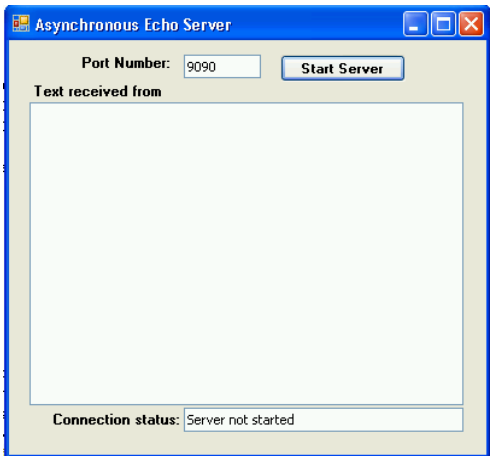
Asynchronous Socket & Stream Methods …

- **Asynchronous** methods of the **Stream** class:

IAsyncResult **BeginRead**(**byte**[] buffer, **int** offset, **int** count, AsyncCallback callback, object state) **int** **EndRead**(IAsyncResult asyncResult)
IAsyncResult **BeginWrite**(**byte**[] buffer,**int** offset,**int** count, AsyncCallback callback, object state) void **EndWrite**(IAsyncResult asyncResult)

Example

- The Asynchronous version of the Echo server.



Example …

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

namespace AsyncEchoServer
{
    public partial class Form1 : Form
    {
        private Socket server;
        private Socket client;
        private byte[] data = new byte[1024];

        public Form1() {
            InitializeComponent();
            Control.CheckForIllegalCrossThreadCalls = false;
        }
```

Example …

```
18.  private void btnStart_Click(object sender, EventArgs e) {
19.      server = new Socket(AddressFamily.InterNetwork,
20.                  SocketType.Stream, ProtocolType.Tcp);
21.      int port = int.Parse(txtPort.Text);
22.      IPEndPoint localEP = new IPEndPoint(IPAddress.Any, port);
23.      server.Bind(localEP);
24.      server.Listen(4);
25.      server.BeginAccept(new AsyncCallback(OnConnected), null);
26.      btnStart.Enabled = false;
27.      txtStatus.Text = "Waiting for Client...";
28.  }
29.  void OnConnected(IAsyncResult result) {
30.      try {
31.          client = server.EndAccept(result);
32.          txtStatus.Text = "Connected to: " + client.RemoteEndPoint;
33.          byte[] message = Encoding.ASCII.GetBytes("Welcome to my Server");
34.          client.BeginSend(message, 0, message.Length, SocketFlags.None,
35.                          new AsyncCallback(OnDataSent), null);
36.      }
37.      catch (SocketException) {
38.          CloseClient();
39.      }
40.  }
```


Example …

```
41.    void OnDataSent(IAsyncResult result) {
42.      try  {
43.         int sent = client.EndSend(result);
44.         client.BeginReceive(data, 0, data.Length,
45.         SocketFlags.None, new AsyncCallback(OnDataReceived), null);
46.       }
47.       catch (SocketException)
48.       {
49.           CloseClient();
50.       }
51.    }

52.    public void CloseClient()
53.    {
54.       client.Close();
55.       txtStatus.Text = "Waiting for Clients...";
56.       server.BeginAccept(new AsyncCallback(OnConnected), null);
57.    }
58.
```

Example …

```
59.  void OnDataReceived(IAsyncResult result) {
60.      try {
61.          int receive = client.EndReceive(result);
62.          if (receive == 0) {
63.              CloseClient();
64.              return;
65.          }
66.          else {
67.              string message = Encoding.ASCII.GetString(data, 0,
68.                                          receive);
69.              lstResult.Items.Add(message);
70.              byte[] echoMessage = Encoding.ASCII.GetBytes(message);
71.              client.BeginSend(echoMessage, 0, echoMessage.Length,
72.                    SocketFlags.None,
73.                    new AsyncCallback(OnDataSent), null);
74.          }
75.      }
76.      catch (SocketException) {
77.          CloseClient();
78.      }
79.  }
```


Example …

- The Asynchronous version of the Echo client.

Example …

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

namespace AsyncEchoClient
{
    public partial class Form1 : Form
    {
        private byte[] data = new byte[1024];
        private Socket client;

        public Form1()
        {
            InitializeComponent();
            Control.CheckForIllegalCrossThreadCalls = false;
        }
```


Example …

```
private void btnConnect_Click(object sender, EventArgs e)
{
    try
    {
        btnConnect.Enabled = false;
        btnDisconnect.Enabled = true;
        btnSend.Enabled = true;

        txtStatus.Text = "Connecting...";
        client = new Socket(AddressFamily.InterNetwork,
                            SocketType.Stream, ProtocolType.Tcp);
        int port = int.Parse(txtPort.Text);
        IPEndPoint remoteEP = new IPEndPoint(IPAddress.Parse(txtIP.Text),
                                             port);
        client.BeginConnect(remoteEP, new AsyncCallback(OnConnected),
                                                   null);
    }
    catch (SocketException)
    {
        CloseConnection();
    }
}
```

Example …

```
30.  void OnConnected(IAsyncResult result) {
31.      try  {
32.          client.EndConnect(result);
33.          txtStatus.Text = "Connected to: " + client.RemoteEndPoint;
34.          client.BeginReceive(data, 0, data.Length, SocketFlags.None,
35.                              new AsyncCallback(OnDataReceived), null);
36.      }
37.      catch (SocketException) {
38.          CloseConnection();
39.      }
40.  }

41.  void OnDataReceived(IAsyncResult result) {
42.      try  {
43.          int receive = client.EndReceive(result);
44.          string message = Encoding.ASCII.GetString(data, 0, receive);
45.          lstResult.Items.Add(message);
46.      }
47.      catch (Exception) {
48.          CloseConnection();
49.      }
50.  }
```


Example …

```
51.  private void btnSend_Click(object sender, EventArgs e){
52.      try {
53.          byte[] message = Encoding.ASCII.GetBytes(txtMessage.Text);
54.          txtMessage.Clear();
55.          client.BeginSend(message, 0, message.Length, SocketFlags.None,
56.                           new AsyncCallback(OnDataSent), null);
57.      }
58.      catch (SocketException) {
59.          CloseConnection();
60.      }
61.  }

62.  void OnDataSent(IAsyncResult result) {
63.      try  {
64.          int sent = client.EndSend(result);
65.          client.BeginReceive(data, 0, data.Length, SocketFlags.None,
66.                              new AsyncCallback(OnDataReceived), null);
67.      }
68.      catch (SocketException) {
69.          CloseConnection();
70.      }
71.  }
```

Example …

```
72.      private void btnDisconnect_Click(object sender, EventArgs
73.      {
74.         CloseConnection();
75.      }
76.
77.      public void CloseConnection()
78.      {
79.         client.Close();
80.         txtStatus.Text = "Disconnected";
81.         btnConnect.Enabled = true;
82.         btnDisconnect.Enabled = false;
83.         btnSend.Enabled = false;
84.      }
85.   }
86. }
```


User-Defined Asynchronous Methods

- Asynchronous methods are useful in writing responsive programs
  - Especially when the operation being done will take long time or is beyond the control of the application.
- Can we write our own asynchronous methods?
  - Yes, In .NET, one can call any method asynchronously.
- Asynchronous support is provided through delegates
- When you compile a delegate, the system automatically adds **BeginInvoke** and **EndInvoke** methods that you can call to execute the referenced method asynchronously
- These **BeginInvoke** and **EndInvoke** methods have similar signature and behaves exactly as the Begin and End methods of the Socket class

Example

- Suppose we wish to call a method with the following signature asynchronously:

  int LongOperation(int param)

- We define a delegate with similar signature:

  delegate int LongOperationDelegate(int aParam);

- The compiler automatically generates and adds the following **BeginInvoke** and **EndInvoke** methods to the delegate:

  IAsyncResult BeginInvoke(int para, AsyncCallback cb, Object state);

  int EndInvoke(IAsyncResult ar);

- **BeginInvoke** takes two more parameters in addition to those used by the method: **AsyncCallback** and **Object**. It also returns **IAsyncResult.**
- The parameters for **EndInvoke** are any *ref* or *out* parameters of the method, followed by an IAsyncResult parameter. It also returns the same type as the method.

Example …

- User-Defined Synchronous Methods

```
using System;

public class UserDefinedAsync {
  delegate int LongOperationDelegate(int time, ref int refParam);

  public static void Main() {
     LongOperationDelegate dlgt = new
                                  LongOperationDelegate(LongOperation);
     int refParam = 0;
     IAsyncResult ar = dlgt.BeginInvoke(5000, ref refParam,
                                 new AsyncCallback(OnCompletion), dlgt);
      // do some work while waiting for completion
     while (!ar.IsCompleted) {
        Console.WriteLine("Waiting for completion");
        System.Threading.Thread.Sleep(1000);
     }
    }
```

Example …

```
16.    public static void OnCompletion(IAsyncResult ar) {
17.       int refParam = 0;
18.       LongOperationDelegate dlgt =
19.                      (LongOperationDelegate) ar.AsyncState;
20.       int longResult = dlgt.EndInvoke(ref refParam, ar);
21.       Console.WriteLine("longResult={0}", longResult);
22.       Console.WriteLine("refParam={0}", refParam);
23.       Console.WriteLine("Press Enter to exit the program.");
24.       Console.ReadLine();
25.    }
26.    public static int LongOperation(int time, ref int refParam) {
27.       while (time > 0) {// simulate a long-running operation
28.         Console.WriteLine("*LongOperation");
29.         System.Threading.Thread.Sleep(1000);
30.         time -= 1000;
31.       }
32.       refParam = 42;
33.       return 1;
34.     }
35.  }
```

```
Waiting for completion
*LongOperation
Waiting for completion
*LongOperation
Waiting for completion
*LongOperation
Waiting for completion
*LongOperation
Waiting for completion
*LongOperation
Waiting for completion
longResult=1
refParam=42
Press Enter to exit the program.
```


Objectives

- Part 1:
  - Understand **asynchronous** methods of the **Socket** & **Stream** classes
  - Create **Asynchronous** client-server applications
  - Write User-Defined Asynchronous Methods
- Part 2:
  - Create multi-client TCP servers using **Thread** class
  - Create multi-client TCP servers using **ThreadPool** class
  - Create multi-client TCP servers using **asynchronous methods**

Multi-Client Servers

- A Multi-Client Server is one that interacts with more than one client at the same time.
- Most real-life servers such as, Web server, Mail server, etc., are multi-client servers
- In UDP, since there is no connection, every server is multi-client server.
- However, in TCP, especially where connection must be maintained throughout a session, a multi-client server must be specifically designed.
- In C#, this can be achieved using the Thread class, the ThreadPool class or by using Asynchronous methods.

Multi-Client Server using Thread

- Threads can be used to make a server that maintains multiple sessions – with multiple clients.
- To a achieve this, the server is partitioned into two parts: connection acceptor and connection handler.
- The connection acceptor is the main application that runs in an infinite loop performing the following tasks:
  - listens for the next client
  - when a client connects, creates an instance of the "connection handler"
  - creates a thread, associate it with the connection handler instance
  - starts the thread to handle communication with this client.
- The connection handler communicates with a client through a method matching the signature of **ThreadStart** delegate

Multi-Client Server using Thread …

```
1.  class MultiThreadedTcpServer {
2.    public static void Main(string[] args) {
3.      int port = 9070;
4.      Socket server = new Socket(AddressFamily.InterNetwork,
5.                                 SocketType.Stream, ProtocolType.Tcp);
6.      IPEndPoint endpoint = new IPEndPoint(IPAddress.Any, port);
7.      server.Bind(endpoint);
8.      server.Listen(10);
9.      Console.WriteLine("Waiting for clients on port " + port);
10.     while(true) {
11.       try {
12.         Socket client = server.Accept();
13.         ConnectionHandler handler=new ConnectionHandler(client);
14.         Thread thread = new Thread(new
15.                     ThreadStart(handler.HandleConnection));
16.         thread.Start();
17.       } catch(Exception) {
18.         Console.WriteLine("Connection failed on port "+port);
19.       }
20.     }
21.   }
22. }
```


Multi-Client Server using Thread …

```
1.  class ConnectionHandler {
2.    private Socket client;
3.    private NetworkStream ns;
4.    private StreamReader reader;
5.    private StreamWriter writer;
6.    private static int connections = 0;
7.    public ConnectionHandler(Socket client) {
8.      this.client = client;
9.    }
10.   public void HandleConnection() {
11.     try {
12.       ns = new NetworkStream(client);
13.       reader = new StreamReader(ns);
14.       writer = new StreamWriter(ns);
15.       connections++;
16.       Console.WriteLine("New client accepted: {0}
17.                         active connections", connections);
18.       writer.WriteLine("Welcome to my server");
19.       writer.Flush();
20.       string input;
```

Multi-Client Server using Thread …

```
21.         while(true) {
22.             input = reader.ReadLine();
23.             if (input.Length == 0 || input.ToLower() == "exit")
24.                     break;
25.             writer.WriteLine(input);
26.             writer.Flush();
27.         }
28.         ns.Close();
29.         client.Close();
30.         connections--;
31.         Console.WriteLine("Client disconnected: {0}
32.                             active connections", connections);
33.       } catch(Exception) {
34.         connections--;
35.         Console.WriteLine("Client disconnected: {0}
36.                             active connections", connections);
37.       }
38.     }
39.  }
```

- The above server can be tested using any of the TCP echo client discussed earlier.

Multi-Client Server using ThreadPool

- Firing a thread to handle each client without any control is a sure way to crash a server.
- Most computers have single processor, which is shared among all the threads.
  - The more the threads, the less the time allocated for each thread, and, the slower the response time of the server.
  - Also creating and destroying threads adds a lot processing overheads to the system.
- To avoid this problems, C# provides the **ThreadPool** class.
- The **ThreadPool** class provides a set of **reusable** threads, maintained by the system, which can be used to assigned tasks normally done by user threads.
- ThreadPool allows a maximum of 25 threads. Any request for a thread is enqueued until threads become available.

Multi-Client Server using ThreadPool …

- Methods of the **ThreadPool** class, all of them static:

bool QueueUserWorkItem (WaitCallback callBack) **bool QueueUserWorkItem** (WaitCallback callBack, object state)	Queues a user method for execution. The method executes when a thread becomes available. The second version can used to pass state information.
void **GetMaxThreads** (**out int** workerThreads, **out int** completionPortThreads)	Returns the maximum number of threads in the pool. All requests above that number remain queued until threads become available.
void **GetAvailableThreads** (**out int** workerThreads, **out int** completionPortThreads)	Returns the difference between the maximum number of threads in the pool, and the number currently active.

Multi-Client Server using ThreadPool …

```
1.  class ThreadPoolTcpServer {
2.    public static void Main(string[] args) {
3.      int port = 9080;
4.      Socket server = new Socket(AddressFamily.InterNetwork,
5.                                 SocketType.Stream, ProtocolType.Tcp);
6.      IPEndPoint endpoint = new IPEndPoint(IPAddress.Any, port);
7.      server.Bind(endpoint);
8.      server.Listen(10);
9.      Console.WriteLine("Waiting for clients on port " + port);
10.     while(true) {
11.       try {
12.         Socket client = server.Accept();
13.         ConnectionHandler handler = new ConnectionHandler(client);
14.         ThreadPool.QueueUserWorkItem(new
15.                          WaitCallback(handler.HandleConnection));
16.       } catch(Exception) {
17.           Console.WriteLine("Connection failed on port "+port);
18.       }
19.     }
20.   }
21. }
```

Multi-Client Server using ThreadPool …

```
class ConnectionHandler {
  private Socket client;
  private NetworkStream ns;
  private StreamReader reader;
  private StreamWriter writer;
  private static int connections = 0;
  public ConnectionHandler(Socket client) {
    this.client = client;
  }
  public void HandleConnection(Object state) {
    try {
      ns = new NetworkStream(client);
      reader = new StreamReader(ns);
      writer = new StreamWriter(ns);
      connections++;
      Console.WriteLine("New client accepted: {0}
                                 active connections", connections);
      writer.WriteLine("Welcome to my server");
      writer.Flush();
      string input;
```

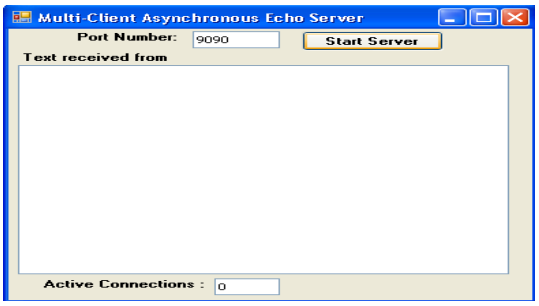

Multi-Client Server using ThreadPool …

```
      while(true) {
          input = reader.ReadLine();
          if (input.Length == 0 || input.ToLower() == "exit")
                break;
          writer.WriteLine(input);
          writer.Flush();
      }
      ns.Close();
      client.Close();
      connections--;
      Console.WriteLine("Client disconnected: {0}
                         active connections", connections);
    } catch(Exception) {
      connections--;
      Console.WriteLine("Client disconnected: {0}
                         active connections", connections);
    }
  }
}
```

- Note: The signature of HandleConnection method has changed to match that of **WaitCallback** delegate.

Multi-Client Asynchronous Server

- The third option in creating multi-client application is by using asynchronous methods.
- The server uses BeginAccept / EndAccept to accept a client.
- Once a client connects and the EndAccept method is called, another call to BeginAccept is made to wait for the next client.
- Communication is initiated with the accepted client using the BeginSend / EndSend methods.



Multi-Client Asynchronous Server …

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

namespace MultiClientAsyncServer
{
    public partial class Form1 : Form
    {
        private Socket server;
        private byte[] data = new byte[1024];
        private int connections = 0;

        public Form1()
        {
            InitializeComponent();
            Control.CheckForIllegalCrossThreadCalls = false;
        }
```

Multi-Client Asynchronous Server …

```
19. private void btnStart_Click(object sender, EventArgs e) {
20.     server = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
21.                         ProtocolType.Tcp);
22.     int port = int.Parse(txtPort.Text);
23.     IPEndPoint localEP = new IPEndPoint(IPAddress.Any, port);
24.     server.Bind(localEP);
25.     server.Listen(4);
26.     btnStart.Enabled = false;
27.     server.BeginAccept(new AsyncCallback(OnConnected), null);
28. }
29. void OnConnected(IAsyncResult result) {
30.     Socket client = server.EndAccept(result);
31.     connections++;
32.     server.BeginAccept(new AsyncCallback(OnConnected), null);
33.     try {
34.         txtStatus.Text = "" + connections;
35.         byte[] message = Encoding.ASCII.GetBytes("Welcome to my Server");
36.         client.BeginSend(message, 0, message.Length, SocketFlags.None,
37.                          new AsyncCallback(OnDataSent), client);
38.     }
39.     catch (SocketException) {
40.         CloseClient(client);
41.     }
42. }
```


Multi-Client Asynchronous Server …

```
19. void OnDataSent(IAsyncResult result)
20. {
21.     Socket client = (Socket)result.AsyncState;
22.     try
23.     {
24.         int sent = client.EndSend(result);
25.         client.BeginReceive(data, 0, data.Length, SocketFlags.None,
26.                             new AsyncCallback(OnDataReceived), client);
27.     }
28.     catch (SocketException)
29.     {
30.         CloseClient(client);
31.     }
32. }
33. public void CloseClient(Socket client)
34. {
35.     client.Close();
36.     connections--;
37.     txtStatus.Text = "" + connections;
38. }
```

Multi-Client Asynchronous Server …

```
19.  void OnDataReceived(IAsyncResult result) {
20.      Socket client = (Socket)result.AsyncState;
21.      try {
22.          int receive = client.EndReceive(result);
23.          if (receive == 0) {
24.              CloseClient(client);
25.              return;
26.          }
27.          else  {
28.              string message = Encoding.ASCII.GetString(data, 0, receive);
29.              lstResult.Items.Add(message);
30.              byte[] echoMessage = Encoding.ASCII.GetBytes(message);
31.              client.BeginSend(echoMessage, 0, echoMessage.Length,
32.                 SocketFlags.None,  new AsyncCallback(OnDataSent), client);
33.          }
34.      }
35.      catch (SocketException)
36.      {
37.          CloseClient(client);
38.      }
39.   }
40.  } }
```


Resources

- MSDN Library
  - Asynchronous Programming Overview
    - http://msdn.microsoft.com/en-us/library/ms228963.aspx
  - Using an Asynchronous Server Socket
    - http://msdn.microsoft.com/en-us/library/5w7b7x5f.aspx
  - Using Asynchronous Sockets
    - http://codeidol.com/csharp/csharp-network/Asynchronous-Sockets/Using-Asynchronous-Sockets/
  - Calling Synchronous Methods Asynchronously
    - http://msdn.microsoft.com/en-us/library/2e08f6yc(VS.80).aspx
- Books
  - Richard Blum, C# Network Programming. Sybex 2002.
- Lecture notes of previous offerings of SWE344 and ICS343
- Some other web sites and books; check the course website at http://faculty.kfupm.edu.sa/ics/alfy/files/teaching/swe344/index.htm