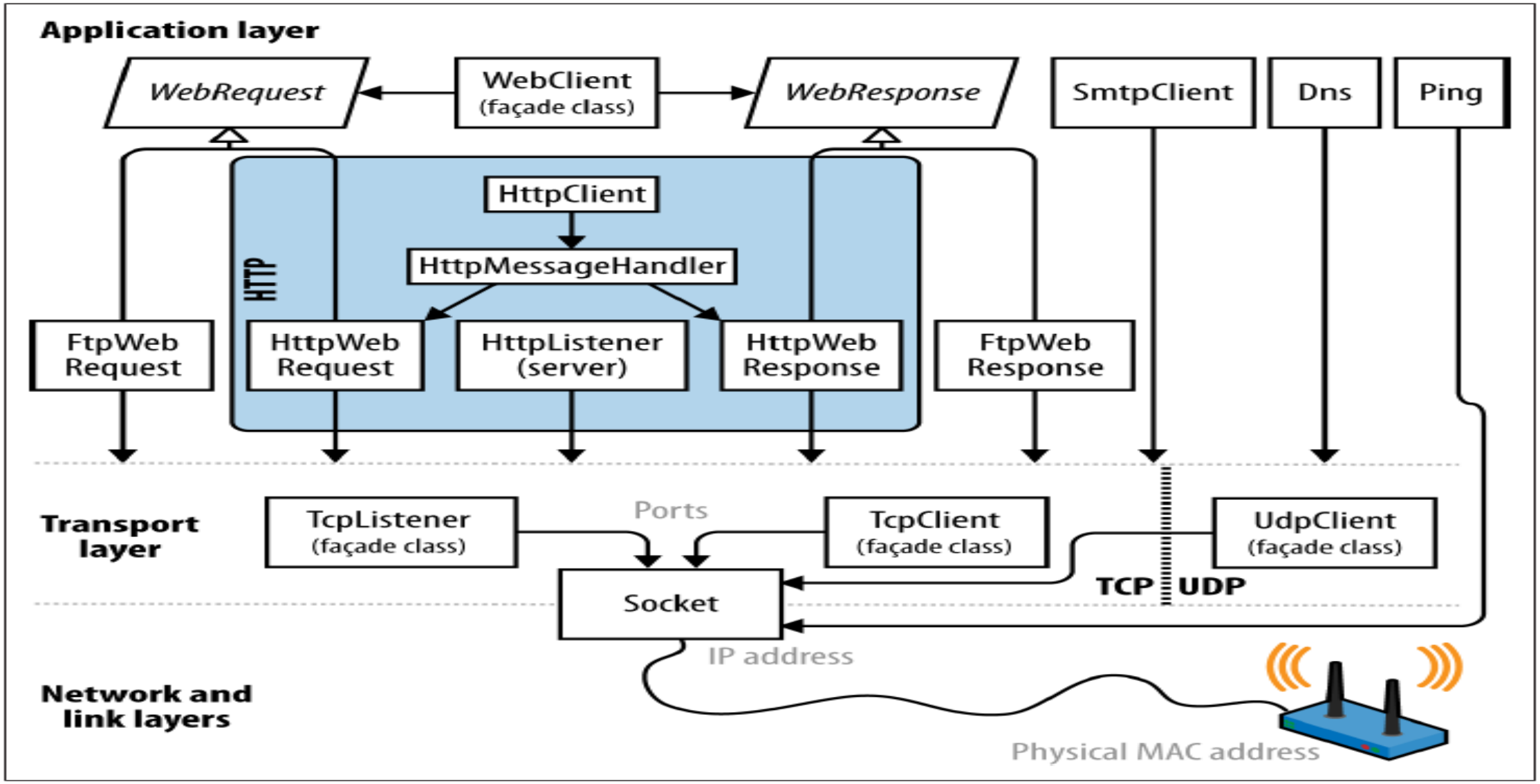
Visual C# .Net using framework 4.5

Eng. Mahmoud Ouf

Lecture 12

Network Architecture



Type of Communications

There are two types of communications:

- ***Connection oriented communications***

  Which are similar to the telephone system, in which a connection is established and held for the length of the session
- ***Connectionless communications***

  Which are similar to the postal service, in which two letters mailed from the same place and to the same destination may actually take two different paths through the system and even arrive at different times

Communication Protocols

There are two types of protocols:

- ***Transmission Control Protocol (TCP)***

  connection-oriented communication protocol which guarantees that sent packets will arrive at the intended receiver undamaged and in the correct sequence.

  If packets of information don't arrive at the recipient, TCP ensures that the packets are sent again.

  If the packets arrive out of order, TCP reassembles them in the correct order transparently to the receiving application.

  If duplicate packets arrive, TCP discards them.

Communication Protocols

- ***User Datagram Protocol (UDP)***

connectionless communication protocol. It incurs the minimum overhead necessary to communicate between applications.

UDP makes no guarantees that packets, called datagrams, will reach their destination or arrive in their original order.

Socket Programming

Addresses and Ports

To communication with computer or device, computer requires an address. The Internet uses two addressing systems:

IPv4 (Currently the dominant addressing system)

IPv4 addresses are 32 bits wide.

When string-formatted, IPv4 addresses are written as four dot-separated decimals (e.g., 101.102.103.104).

An address can be unique in the world or unique within a particular subnet (such as on a corporate network).

IPv6 (The newer 128-bit addressing system)

Addresses are string-formatted in hexadecimal with a colon separator (e.g., [3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]).

Addresses and Ports

The TCP and UDP protocols break out each IP address into 65,535 ports, allowing a computer on a single address to run multiple applications, each on its own port.

Many applications have standard port assignments; for instance, HTTP uses port 80; SMTP uses port 25.

Ports till 1024 are reserved for the Operating System, So use a port number above 1024

Establishing a Simple TCP Server

This is 5 Steps Process:

1) Create an object of class ***TcpListner***, which represents a TCP stream socket through which a server can listen for requests

TcpListner server = new TcpListner(localaddr, port);

Where:

localaddr : is the local address of the Server, object from IPAddress

```
byte []bt = new byte[]{127,0,0,1};
IPAddress localaddr = new IPAddress(bt);
```

port : bind the server application to a specific port. Number above 1024

Establishing a Simple TCP Server

2) Call ***TcpListner***'s ***Start*** method, which makes the ***TcpListner*** object begins to listen for connection request.

   The Server creates a connection to the client when it receives a connection request. This is done by calling ***TcpListner***'s ***AcceptSocket*** method which return object of class ***Socket*** that is used to manage a connection to client

```
server.Start();
Socket connection = server.AcceptSocket();
```

3) Establish the stream used for communication with the client.

   This is done by creating a ***NetworkStream*** object that uses the ***Socket*** object representing the connection to perform the actual sending and receiving data.

```
NetworkStream nStream = new NetworkStream(connection);
```

Establishing a Simple TCP Server

4) Process the data between server/client using (ReadByte/WriteByte), or use ***BinaryReader*** and ***BinaryWriter*** objects.

BinaryReader br = new BinaryReader(nStream);

5) Terminating connection, this is done when the client and server finish communication, the server calls method ***Close*** of ***BinaryReader, BinaryWriter, NetworkStream*** and ***Socket*** to terminate the connection. The server then returns to step two to wait for the next connection request.

Note: it is recommended to call method ***Shutdown*** before method ***Close*** to ensure that all data is sent and received before the Socket closes.

```
br.Close();
nStream.Close();
connection.Shutdown();
connection.Close();
```

Establishing a Simple TCP Client

This is 4 Steps Process:

1) Create an object of class ***TcpClient*** to connect to the server. The connection is established by calling ***TcpClient*** method ***Connect***.
   One overloaded version of this method takes two arguments the server's IP address and its port number as in:

   TcpClient client = new TcpClient();

   client.Connect(serverAddress, serverPort);

   If the connection is successful, TcpClient method Connect returns a positive integer; otherwise, it returns 0.

   NetworkStream nStream = client.GetStream();

   BinaryReader br = new BinaryReader(nStream);

Establishing a Simple TCP Client

2) The ***TcpClient*** uses its ***GetStream*** method to get a ***NetworkStream*** so that it can write to and read from the server.
We then use the ***NetworkStream*** object to create a ***BinaryWriter*** and a ***BinaryReader*** that will be used to send information to and receive information from the server, respectively.

```
NetworkStream nStream = client.GetStream();
BinaryReader br = new BinaryReader(nStream);
```

3) The processing phase, in which the client and the server communicate. In this phase, the client uses ***BinaryWriter*** method ***Write*** and ***BinaryReader*** method ***ReadString*** to perform the appropriate communications.

Establishing a Simple TCP Client

4) After the transmission is complete, the client have to close the connection by calling method Close on each of BinaryReader, BinaryWriter, NetworkStream and TcpClient. This closes each of the streams and the TcpClient's Socket to terminate the connection with the server.

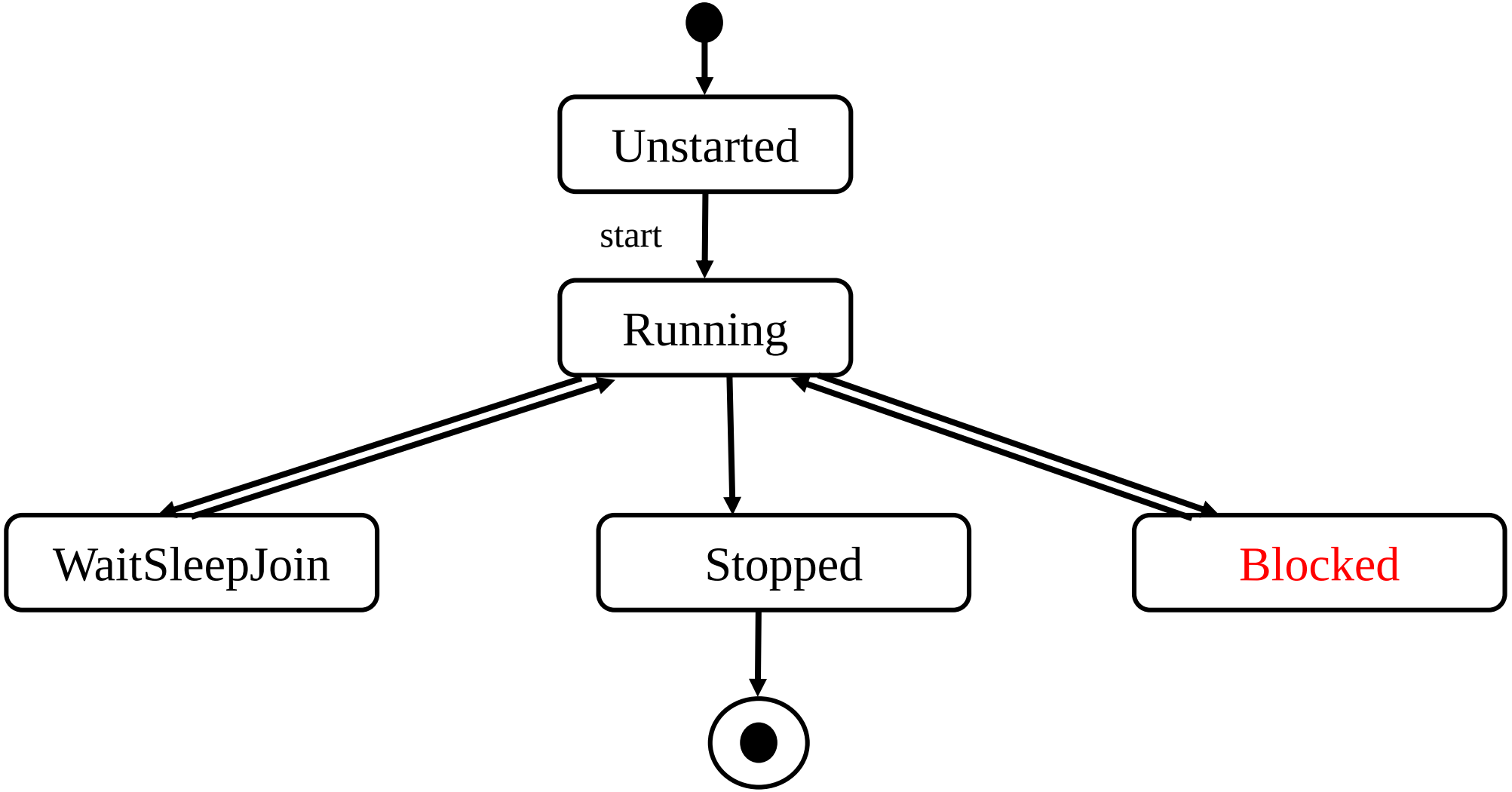
```
br.Close();
nStream.Close();
```

Threading

Introduction

The .NET Framework Class Library provides concurrency primitives.
You specify that applications contain "threads of execution," each of which designates a portion of a program that may execute concurrently with other threads this capability is called multithreading.
Multithreading is available to all .NET programming languages, including C#, Visual Basic and Visual C++.
The .NET Framework Class Library includes multithreading capabilities in namespace System.Threading.
The basic concept behind threading is to perform multiple operations concurrently.
Each of these operations can be thought of a separate thread of logic.
Example: The Garbage Collector in .Net

Thread Life Cycle



Thread Priorities and Thread Scheduling

The Windows operating system supports a concept, called timeslicing, that enables threads of equal priority to share a processor.

Without timeslicing, each thread in a set of equal-priority threads runs to completion (unless the thread leaves the Running state and enters the WaitSleepJoin, Suspended or Blocked state) before the thread's peers get a chance to execute.

With timeslicing, each thread receives a brief burst of processor time, called a quantum, during which the thread can execute.

At the completion of the quantum, even if the thread has not finished executing, the processor is taken away from that thread and given to the next thread of equal priority, if one is available.

On a single-core computer, the Windows operating system allocate “slices” of time to each thread (typically 20 ms in Windows)

Thread Priorities and Thread Scheduling

The job of the thread scheduler is to keep the highest-priority thread running at all times and, if there is more than one highest-priority thread, to ensure that all such threads execute for a quantum in round-robin

Example: assuming a single-processor computer, threads A and B each execute for a quantum in round-robin fashion until both threads complete execution.

This means that A gets a quantum of time to run. Then B gets a quantum. Then A gets another quantum. Then B gets another quantum.

This continues until one thread completes. The processor then devotes all its power to the thread that remains (unless another thread of that priority is started).

Thread Priorities and Thread Scheduling

A thread's priority can be adjusted with the Priority property, which accepts values from the ThreadPriority enumeration
(Lowest, BelowNormal, Normal, AboveNormal and Highest. By default, each thread has priority Normal).

Creating and Executing Threads

To create a thread follow the following steps:

1) Create an object from class Thread. This class has only one constructor takes an object from ThreadStart delegate

2) So, We must define an object from ThreadStart delegate to pass it to the Thread class constructor

3) Before creating object from the delegate, we must define the function that will be attached to that delegate and will execute when the Thread object runs.

This function must : return void and didn’t take any parameter

Note: the Last sequence must be: 3, then 2, then 1

4) Call the Start method of the Thread class to begin execution of the thread.

Creating and Executing Threads

```
class ThreadTest
{
        static void Main()
        {
                Thread t = new Thread (WriteY); // Kick off a new thread
                t.Start(); // running WriteY()
                // Simultaneously, do something on the main thread.
                for (int i = 0; i < 1000; i++) Console.Write ("x");
        }
        static void WriteY()
        {
                for (int i = 0; i < 1000; i++) Console.Write ("y");
        }
}
```