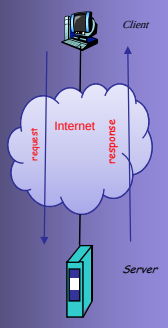

INTERNET PROTOCOLS AND CLIENT-SERVER PROGRAMMING SWE344

Fall Semester 2008-2009 (081)

Module 6.2: C# UDP C/S Programming (Part 2)

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

Objectives

- Learn about advantages of UDP applications
- Learn about problems associated with UDP applications and how to handle them
- Learn how to control the behavior of a Socket using Socket Options
- Learn how to catch SocketException and find the cause of the error

Advantages of UDP Applications

Advantage 1: Speed

- Because no connection/disconnection overheads, no buffering and other error checks, UDP packets gets to their destination (if they do) much faster.
  - For certain applications speed is more important than reliability – e.g multimedia applications.

Advantage 2: Message Boundary makes for easier programming

- UDP preserves message boundaries between clients and servers.
  - No internals buffers.
    - Thus, a UDP packet sent using one call to SendTo () is directly received using one call to ReceiveFrom()
    - This 1-1 correspondence makes programming a UDP client-server applications easier than TCP

Example 1

- The following shows the 1-1 correspondence between calls to **SendTo()** by a sender and calls to the **ReceiveFrom()** by a receiver:

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

class UdpBoudaryTestServer
{
   public static void Main()
   {
      IPEndPoint localEP = new IPEndPoint(IPAddress.Any, 9050);
      Socket server = new Socket(AddressFamily.InterNetwork,
                        SocketType.Dgram, ProtocolType.Udp);
      server.Bind(localEP);
      Console.WriteLine("Waiting for a client...");
```

Example 1 …

```
14.         //dummy end-point
15.         EndPoint remoteEP = new IPEndPoint(IPAddress.Any, 0);
16.
17.         byte[] data;
18.         int recv;

19.         for(int i = 0; i < 5; i++)
20.         {
21.            data = new byte[1024];
22.            recv = server.ReceiveFrom(data, ref remoteEP);
23.            Console.WriteLine("Received message from {0}: {1}",
24.            remoteEP,Encoding.ASCII.GetString(data, 0, recv));
25.         }
26.         server.Close();
27.      }
28.  }
```


Example 1 …

```
1.   using System;
2.   using System.Net;
3.   using System.Net.Sockets;
4.   using System.Text;
5.   class TestUdpClient
6.   {
7.      public static void Main()
8.      {
9.         IPEndPoint remoteEP = new IPEndPoint(
10.                        IPAddress.Parse("127.0.0.1"), 9050);
11.        Socket client = new Socket(AddressFamily.InterNetwork,
12.                       SocketType.Dgram, ProtocolType.Udp);
13.       for (int i=0; i<5; i++)
14.          client.SendTo(Encoding.ASCII.GetBytes("Message "+i),
15.                                                  remoteEP);
16.       Console.WriteLine("Messages sent");
17.       Console.WriteLine("Stopping client");
18.       client.Close();
19.     }
20.  }
```

Problems with UDP Applications

- In the previous example, the server makes 5 calls to the **ReceiveFrom** method in a loop
- Possible problems are:
  - What if the client sends less than five messages?
  - Also what if some of the messages got lost before they reach the server?
- Since the **ReceiveFrom()** is a blocking method, the server will be waiting indefinitely for the next packet.
- A solution to this problem is to use **SetSocketOption()** to set a receive time-out for the socket.

Socket Options

- Various Socket options can be set using SetSocketOption() of the Socket class
- This method is overloaded:

  SetSocketOption (SocketOptionLevel *sl*, SocketOptionName *sn*, byte[] *value*)
  SetSocketOption (SocketOptionLevel *sl*, SocketOptionName *sn*, int *value*)
  SetSocketOption (SocketOptionLevel *sl*, SocketOptionName *sn*, object *value*)
  SetSocketOption (SocketOptionLevel *sl*, SocketOptionName *sn*, bool *value*)

- It allows different options to be modified at the level of the socket itself or the underlying protocols (UDP, TCP or IP).

Socket Options …

- The **SocketOptionLevel** is an enumeration with the following possible values:

Value	Description
Socket	Socket Level Option
Udp	Udp Level Option
Tcp	Tcp Level Option
IP	IP Level Option

Socket Options …

- **SocketOptionName** is an enumeration that specifies the name of the Socket Option
- Some SocketOptionNames and their SocketOptionLevel

ReceiveTimeout	Socket	Sets the Receives time-out value
SendTimeout	Socket	Sets Sends time-out value
ReceiveBuffer	Socket	Sets the receive buffer size
SendBuffer	Socket	Sets the send buffer size
Broadcast	Socket	Permits sending broadcast messages
NoChecksum	Udp	Sends UDP packets with checksum as 0
AddMembership	IP	Sets an IP to Join Multicast Group
DropMembership	IP	Sets an IP to Drop from Multicast Group
MulticastTimeToLive	IP	Sets the IP multicast time to live

Socket Options …

- The third parameter specifies a value for the SocketOption
- Depending on the option being set, the value can be of type int, byte array, bool or an object – hence the four versions.
- The Socket class also has **GetSocketOption** method that can be used to get the current settings for a socket option.

```
object GetSocketOption(SocketOptionLevel sl,  SocketOptionName sn)
void GetSocketOption(SocketOptionLevel sl, SocketOptionName sn,
                     byte[] value)
```

- For options returning int, bool or object, the first version is used, while, the second is used for options returning byte[].
- Note: .NET 2.0 provides many properties that can be used to directly set socket options including: ReceiveTimeout, Ttl, SendTimeout, ReceiveBufferSize, SendBufferSize EnableBroadcast, etc.

Problems with UDP Applications …

- To solve the problem of having ReceiveFrom() blocking indefinitely, we set the **ReceiveTimeOut** option of the socket
  - Its value is an integer representing the time to wait in milliseconds
- The following code sets the server socket to give up after waiting for 10 seconds:

```
server.SetSocketOption(SocketOptionLevel.Socket,
                       SocketOptionName.ReceiveTimeout, 10000);
// Or in .Net 2.0
server.ReceiveTimeout = 10000;
```

- After the 10 seconds, a SocketException is thrown, thus the call to the ReceiveFrom should be inside a try block.

Problems with UDP Applications …

- Another Problem: data can be lost due to small buffer or network error
  - Both **ReceiveFrom()** and **SentTo()** use a byte array to receive and send data
  - Since there is no buffering, if the array is too small to receive the packet, part of the data will be lost and a SocketException will be raised
- **Solution:** Catch the SocketException, increase the array size and request the sender to re-transmit.
- The SocketException has a property, **ErrorCode** that returns int value corresponding to the actual error that caused the Exception
- For a complete list of error codes, refer to the MSDN documentation on the following link:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/windows_sockets_error_codes_2.asp

Problems with UDP Applications …

- Following are examples of some ErrorCode values:

10040	Message too long – Receiving buffer is too small
10041	Protocol wrong type for socket
10048	Address already in use
10050	Network is down
10051	Network is unreachable
10054	Connection reset by peer – when Receive method times out.
10055	No buffer space available
10056	Socket is already connected
10057	Socket is not connected
10058	Cannot send after socket shutdown
10065	Host is unreachable

Example 2

- The following shows how to use the **ReceiveTimeout** option and how to use the **SocketException** to take care of too small data buffer problem.

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

class BestUdpClient
{
   private static byte[] data = new byte[1024];
   private static int size = 10;

    private static int AdvancedSendReceive(Socket s, byte[]
                   message, EndPoint remoteEP)
   {
      int recv = 0;
      int retry = 0;
```


Example 2 …

```
      while (true){
         Console.WriteLine("Attempt #{0}", retry);
         try {
            s.SendTo(message, message.Length, SocketFlags.None,
                     remoteEP);
            data = new byte[size];
            recv = s.ReceiveFrom(data, ref remoteEP);
             return recv;
         } catch (SocketException e) {
            if (e.ErrorCode == 10054) //receive timeout
               recv = 0;
            else if (e.ErrorCode == 10040) { //buffer too small
               Console.WriteLine("Error receiving packet");
               size += 10;
               recv = 0;
            }
         } //catch
```

Example 2 …

```
32.         if (recv == 0) {
33.               retry++;
34.               if (retry > 4)
35.                  return 0;
36.            }
37.         } //while
38.      } // AdvancedSendReceive

39.      public static void Main()  {
40.         IPEndPoint remoteEP = new
41.                   IPEndPoint(IPAddress.Parse("127.0.0.1"), 9050);

42.         Socket client = new Socket(AddressFamily.InterNetwork,
43.                         SocketType.Dgram, ProtocolType.Udp);
44.         client.SetSocketOption(SocketOptionLevel.Socket,
                     SocketOptionName.ReceiveTimeout, 5000);

45.         string input, echo;
46.         int recv;
```


Example 2 …

```
47.          while(true) {
48.            Console.Write("Enter message for the server: ");
49.            input = Console.ReadLine();
50.            if (input == "exit")
51.               break;
52.            recv = AdvancedSendReceive(client,
53.                       Encoding.ASCII.GetBytes(input), remoteEP);
54.            if (recv > 0) {
55.               echo = Encoding.ASCII.GetString(data, 0, recv);
56.               Console.WriteLine("Received from server: " + echo );
57.            } else
58.               Console.WriteLine("Did not receive an answer");
59.         }
60.         Console.WriteLine("Stopping client");
61.         client.Close();
62.      } // main
63.   } // BestUdpClient
```

- Note, this example client can be tested using the **UdpSocketServer**

Do not convert UDP to TCP

- It is obvious that many things can go wrong with UDP transmission.
- Unfortunately, UDP does not have any internal support for handling such problems.
- It is possible for one to programmatically implement some error checking mechanism to solve some of these problems.
- However, overdoing this will clearly defeats the purpose of UDP.
  - Such error-checking mechanisms will require re-transmission and other overheads which could slow-down the UDP transmission.
- Therefore as a rule, if your application requires reliability, just go for TCP instead of UDP.

Resources

- MSDN Library
  - http://msdn.microsoft.com/en-us/default.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