Lab Session 2: TCP / UDP

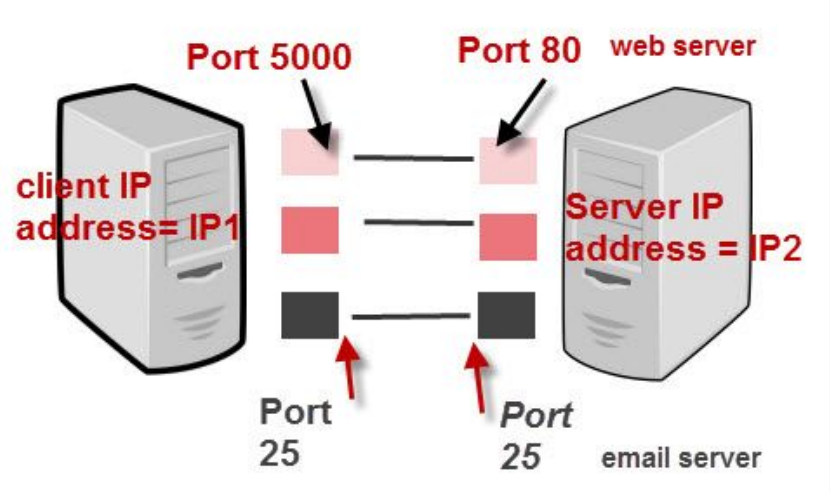
Sockets

Sockets are endpoints in the communication between two processes in the same or different machines.

They use the same actions as any other I/O functions: Read and write

Two main types for this course: TCP and UDP

Need an Address and a Port



Sockets

```
newSocket = new Socket(AddressFamily.InterNetwork,
                       SocketType.Dgram,
                       ProtocolType.Udp);
```

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

InterNetwork -> IPv4 family of addresses
InterNetworkV6 -> IPv6 family of addresses

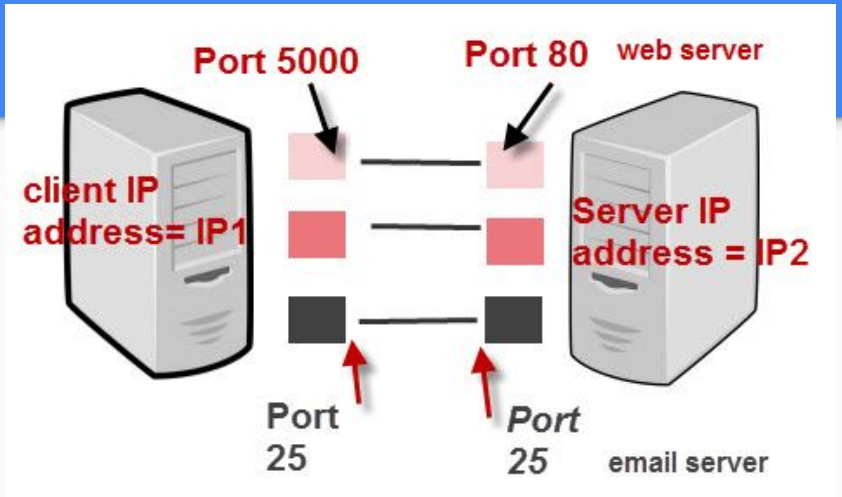
Dgram vs. Stream -> independent packets (Datagrams) vs continuous stream of data

Dgram always for UDP.
Stream always for TCP.

newSocket.Shutdown() -> shutdown the transfer of data, but keep socket alive

newSocket.Close() -> Stop transfers and **destroy** socket.

Addresses



We need an Address to connect two processes!

IP:

- String
- Used for source and destination of data
- IPAddress.Any means any IP address is valid

Port:

- Int
- Used for connecting to each application

Usage:

```
IPEndPoint ipep = new IPEndPoint(IPAddress.Any,port);
or
IPEndPoint ipep = new IPEndPoint(IPAddress.Parse(“192.168.1.45”),port);
```

Binding

Binding is the process of assigning an IP Address to a Socket.

```
newSocket.Bind(ipep);
```

Binding is only necessary to receive data from arbitrary hosts (otherwise, the socket doesn’t need to know!)

Sometimes, if a socket was not properly closed, calling Bind with the same IP/Port will return a SocketError indicating that the address is already in use.

UDP Sockets

`data`: bytes[] Data to receive or be sent

`Remote`: EndPoint, destination or source of the data

`recv`: Int, size of the data

`SocketFlags.None`: Special options. leave as None, for now

```
recv = newSocket.ReceiveFrom(data,ref Remote);
```

```
newSocket.SendTo(data,recv, SocketFlags.None, Remote);
```

https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.socket.receivefrom?view=net-5.0
https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.socket.sendto?view=net-5.0

Remember:

- UDP does not check if sent packets are received
- In UDP, no guarantee that the order of packets sent is preserved at destination
- SendTo and ReceiveFrom are blocking functions

TCP Sockets (1/2)

```
newSocket.Listen(10);
```

Listen(Backlog): Sets the socket in listening mode, and accepts up to N connections

```
client = newSocket.Accept()
```

Accept(): returns a client socket that has tried to connect. Returns error if used in non-blocking mode with no pending connections

Listen and Accept are done on the server side.

```
newSocket.Connect(ipep);
```

Connect(ipEndPoint): tries to connect to a specific endpoint (ipep). Is blocking.

Remember:

- Avoid blocking functions in the main thread
- Handle errors so that your application runs smoothly
- Close threads and connections to avoid problems later
- Can you think of ways to handle multiple connections?

https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.socket.accept?view=net-5.0#System_Net_Sockets_Socket_Accept
https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.socket.listen?view=net-5.0
https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.socket.connect?view=net-5.0

TCP Sockets (2/2)

```
client.Send(data, recv, SocketFlags.None);
```

```
recv = client.Receive(data);
```

Arguments are the same as in UDP

`Receive` returns error when:

- `Socket.ReceiveTimeout` or `Socket.SendTimeout` expired
- No connection available (handshake was interrupted or never started)
- Data is larger than buffer

`Receive` returns 0 if client is disconnected
Both return #bytes sent/received

It's the dev's responsibility to ensure that the all commands are sent safely (check connection, check what the message means, check what the error means...)

https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.socket.send?view=net-5.0
https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.socket.receive?view=net-5.0

TCP vs. UDP

TCP ensures connection and messages are consistent at all times

TCP reads the messages in order of arrival (blocking)

UDP eliminates previous messages from the buffer when a new one arrives

UDP doesn’t use connections neither handshakes (less safe)

UDP typically used when you need the most updated data available

TCP typically used when you need reliable communication between host and client

TO DO’s

TCP edition

TO DO 1

- **Create and bind the socket**

Any IP that wants to connect to the port 9050 using TCP will communicate with this socket

Don't forget to set the socket in listening mode

```
void Socket(AdressFamily addressFamily, SocketType socketType, ProtocolType protocolType);
public void Bind(EndPoint localEp);
socket.Listen(10);
```

TO DO 2

- **Connect the client**

Create the server endpoint so we can try to connect to it. You'll need the server's IP and the port we binded it to before

When calling connect and succeeding, our server socket will create a connection between this endpoint and the server's endpoint

```
public IPEndPoint(IPAddress address, int port);
public void Connect(EndPoint remoteEP);
```

TO DO 3

TCP makes managing connections easy, so we are going to put it to use.

Accept any incoming clients and store them in a user. When accepting, we can now store a copy of our server socket who has established a communication between a local endpoint (server) and the remote endpoint(client)

```
public Socket Accept();
```

TO DO 4

Using the socket that stores the connection between the 2 endpoints, call the TCP send function with an encoded message;

```
public int Send(byte[] buffer);
```

Conversion string-bytes / bytes-string

```
data = Encoding.ASCII.GetBytes(text);
```

```
text = Encoding.ASCII.GetString(data,0,recv);
```

TO DO 5

- **Receiving messages**

Create an infinite loop to start receiving messages for this user

You'll have to use the socket method receive() to be able to get them.

```
public int Receive(byte[] buffer);
```

TO DO 6

- **Answering a with the server**

We'll send a ping back every time a message is received

Start another thread to send a message, same parameters as this one.

We'll use the socket we stored as a user to send a "ping". Just call the same send function as before and encode the string “ping”

TO DO 7

- **Receiving the ping on the client**

Similar to what we already did with the server, we have to call the Receive() method from the socket.

TO DO’s

UDP edition

TO DO 1

- **Create and bind the socket**

UDP doesn't keep track of our connections like TCP. This means that we "can only" reply to other endpoints /since we don't know where or who they are.

We want any UDP connection that wants to communicate with 9050 port to send it to our socket. So as with TCP, we create a socket and bind it to the 9050 port.

```
public IPEndPoint(IPAddress address, int port);
void Socket(AdressFamily addressFamily, SocketType socketType, ProtocolType protocolType);
public void Bind(EndPoint localEp);
```

TO DO 2

- **Send a message to the server**

Unlike with TCP, we don't "connect" first, we are going to send a message to establish our communication so we need an endpoint

We have to use the server's IP and the port we've binded it to. initialize the endpoint with the server information so we can connect to it.

```
public IPEndPoint(IPAddress address, int port);
void Socket(AdressFamily addressFamily, SocketType socketType, ProtocolType protocolType);
```

TO DO 2.1

- **Send a message to the server**

Create a new socket with the correct protocol to use it to send the Handshake to the server's endpoint. This time, our UDP socket doesn't have that information, so we have to pass it as a parameter on it's SendTo() method

```
public int SendTo(byte[] buffer, int size, SocketFlags socketFlags, EndPoint remoteEP);
```

TO DO 3

- **Receive the message and manage the connection**

  We don't know who may be communicating with this server, so we have to create an endpoint with any address and a remote so we can reply to it later. Use receivefrom to block the thread until receiving the message,

```
IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
EndPoint Remote =(EndPoint)(sender );
public int ReceiveFrom(byte[] buffer, ref EndPoint remoteEP);
```

TO DO 4

- **Answering a with the server**

When our UDP server receives a message from a random remote, it has to send a ping.

Use socket.SendTo to send it using the remote we stored earlier.

```
public int SendTo(byte[] buffer, int size, SocketFlags socketFlags, EndPoint remoteEP);
```

TO DO 5

- **Receive the ping**

Same as in the server, in this case the remote is a bit useless, since we already know it's the server who's communicating with us

Other Useful Material

Working Locally

We can work executing both processes in the same machine, referring to itself with the IP address 127.0.0.1 (localhost).

You will have to launch two instances of the application (the client, and the server) to make the test.

Working in Class

Your computers should have their IPs written on their cover.

To know your IP, run “ipconfig” in a terminal, and choose the connection that you are using.

From any computer you can check if you could connect to any other computer by running “ping 192.0.0.0” (where 192.0.0.0 should be changed by your target IP).

Beware of the firewalls: if connection about 2 computers doesn’t work, but it works between two other computers, re-configure the Windows Firewall.

Try-Catch example

Try: Attempt something unless an error is raised

Catch: If the previous try block fails, don't stop the program, but do something else (i.e. fail gracefully)

Usually, print the error (`System.Exception e`) and do something to make sure nothing is "broken" because of the previous failure.

Remember: If the "Catch" block executes, nothing in the "Try" block will have executed (no variables created/modified, no functions run)

```
try
{
    newSocket.Listen(10);
    Debug.Log("Waiting for clients...");
    client = newSocket.Accept();
    clientep = (IPEndPoint)client.RemoteEndPoint;
    Debug.Log("Connected: "+clientep.ToString());
    connected=true;
}
catch (System.Exception e)
{
    Debug.Log("Connection failed.. trying again...");
}
```

Working Remotely

If you want to connect with a remote machine (a classmate's machine), you will have to copy your application to the other machine.

If you need to know your public IP address (your router's IP address publicly visible in the internet), just type "my ip address" in google.

Of course you will not be interested in sending any data packets to your classmate's router, but his/her machine in the local network. You will have to configure your router to forward packets coming into a certain port number to the desired machine. (of your choice, but some high number such as 9999 for instance). Different routers have different interfaces, you will have to investigate on how to do that. Here's some reference:

https://www.noip.com/support/knowledgebase/general-port-forwarding-guide/

Spoiler Alert: We will see how to overcome the problems derived from the NAT layer in your Router, and they exist, in the future.

Alternative TCP Server in Unity

Create the socket `server = new TcpListener(IPAddress.Any,port);`

Stream structures for I/O

```
private StreamReader reader;
private StreamWriter writer;
```

Start the Listener: `server.Start();`

```
private void StartListening()
{
    server.BeginAcceptTcpClient(AcceptTcpClient, server);
}
```

Do not use for this exercise!

`AcceptTcpClient` is any function you want to execute when a new client connects to the server. See this for more information.

Alternative TCP Server in Unity

The server can send back messages to the client with:

```
// create a writer for the client stream
StreamWriter writer = new StreamWriter(c.tcp.GetStream());
// Write the data in the stream buffer
writer.WriteLine(data);
// clear buffer and send data to the stream
writer.Flush();
```

Where “c” is an object containing a connected client, obtained via:

```
new ServerClient(listener.EndAcceptTcpClient(ar))
```

(documentation)

Do not use for this exercise!

Alternative TCP Client in Unity

Define host and application port:

```
string host = "127.0.0.1";
int port = 1234;
```

Open socket:

```
socket = new TcpClient(host, port);
stream = socket.GetStream();
writer = new StreamWriter(stream);
reader = new StreamReader(stream);
```

Read data and do something:

```
if (SocketReady)
{
    if (stream.DataAvailable)
    {
        string data = reader.ReadLine();
        if (data !=null)
        {
            OnIncomingData(data);
        }
    }
}
```

Do not use for this exercise!

Examples and other documentation

Client / Server simple examples:

http://www.java2s.com/Code/CSharp/Network/SimpleUdpClient.htm

http://www.java2s.com/Code/CSharp/Network/SimpleUdpServer.htm

http://www.java2s.com/Code/CSharp/Network/SimpleTcpClient.htm

http://www.java2s.com/Code/CSharp/Network/SimpleTcpServer.htm

Packages used:

```
using System;

using System.Net;

using System.Net.Sockets;

using System.Text;

using System.Threading;
```