NETWORKING LAB 03

HANDS-ON

In the hands-on networking unit the student will start off looking at a HLAPI (High-Level API) which falls within layers 4 and 5 on the TCP/IP generic model. The student will be mostly protected from networking errors, and will get a grasp on how connections can be created between computers. The foundational knowledge of a server authorative networking state is also established, as well as client-side prediction and latency compensation.

The student will then transition into a LLAPI (Low-Level API) which resides within layer 2 and 3 on the TCP/IP generic model. The student will lose a significant portion of the protection provided from the HLAPI as they begin to work directly with socketing and commands.

Section Three: Transportation

In this section of the networking unit, the student will begin to dabble in the LLAPI of unity's networking. This will allow the student to learn basic socket management and message sending over a network. The theory in this lab can carry over to any networking solution.

STEP ONE: SETUP

For the setup of this lab you will need to create a new unity project.

After creating a new unity project, you will need to duplicate the Main Camera in the default scene.

Name one Main Camera "Client" and the other Main Camera "Server".

Finally, create two C# scripts: ServerConnection and ClientConnection.

STEP TWO: STARTING A SERVER

The first thing you will do is create a basic server socket. This will open a socket on the Transport layer of your network protocol to allow connections from other peers to your computer.

To start, open the ServerConnection script you made in the setup step of this section.

Since you will be manually serializing data in this script to send across the network, you will need to include the using System.Runtime.Serialization.Formatters.Binary statement. You will also need to include using System.IO, and using UnityEngine.Networking.

```
using UnityEngine;
using System.Collections;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
using UnityEngine.Networking;

public class ServerConnection : MonoBehaviour {

	// Use this for initialization
	void Start () {

	}

	// Update is called once per frame
	void Update () {

	}
}
```

The next thing you will need to do is declare some variables. When you were working with the NetworkManager, you were keeping track of things such as the IP address and the Port of the server and clients. The NetworkManager also had some useful record-keeping variables such as isLocalPlayer, if the server was running, if people were connected to the server, and how many connections the server could maintain.

Since you will not be using the NetworkManager in this lab, you will have to keep track a majority of this information yourself.

```
	int serverSocketID = -1;
	int maxConnections = 10;
	byte unreliableChannelID;
	byte reliableChannelID;
	bool serverInitialized = false;
```

Most of these variables should be somewhat familiar.

The serverSocketID will store a unique number that the server can be identified by. When you start the server, you will acquire this number and store it for use should you ever need to identify the specific server running on a specific application. You initialize this to -1 so you can later check if the server was successfully started.

The maxConnections just notates how many clients can be connected to the server at once.

Unreliable and reliable Channel ID's both store a unique identifier for a channel. A channel is how messages are sent between computers (using UDP and TCP). In essence what these two variables mean is what is the unique identifier you need to use if you want to send a message via a UDP (which is unreliable – remember, UDP can drop messages at any time) or if you want to send a message via the TCP (which is reliable).

Finally, you have a bool that stores if the server is currently running, or if it not.

Now to actually get to starting a server (or rather – opening a socket for a server).

GLOBAL CONFIGURATION

Before you can actually open a socket, there are some housekeeping items you must attend to. The first of these is something called a GlobalConfig. The GlobalConfig is something that your network will rely on to setup certain configurations. Each client connecting to a server must be running the same configuration as the server.

The GlobalConfig stores information such as the maximum packet size that can be sent across the network, how many messages can be placed in the queue to be transmitted, and how often the network will attempt to transmit data.

The most important piece of information that the GlobalConfig stores is something called a ReactorModel. The ReactorModel determines how the network reacts to packets as the arrive on the computers. There are two main types of Reactor Models:

1. Select
    a. Received packets will be handled immediately when they arrive
2. FixedRate
    a. Received packets will be handled periodically, for example every 100ms

```
	// Use this for initialization
	void Start () {

		GlobalConfig globalConfig = new GlobalConfig ();
		globalConfig.ReactorModel = ReactorModel.FixRateReactor;
```

After defining the ReactorModel, you can set how long the model will wait before doing something. For example, how often will a FixRateReactor check and handle packets? Or how often will a SelectReactor check if there are any messages to send?

This is set in something called the ThreadAwakeTimeout. Since you will be using a FixRateReactor in this lab, then you will set the ThreadAwakeTimeout to 10 (meaning that the reactor will process and handle packets every 10ms). If you were using a SelectReactor, then you would want to set the ThreadAwakeTimeout to something a little bigger (say 20), which would mean that the reactor would check for messages to send and process any messages in queue every 20 ms.

```
        // Use this for initialization
        void Start () {

               GlobalConfig globalConfig = new GlobalConfig ();
               globalConfig.ReactorModel = ReactorModel.FixRateReactor;
               globalConfig.ThreadAwakeTimeout = 10;

        }
```

CHANNEL CONFIGURATION

With the global configuration set up, you can now manually assign some channels (how messages are going to be sent across the network – UDP or TCP). For this lab you will have two channels, one channel for UDP and one channel for TCP. When you define these channels, they will return a unique identifier that you can use to access the channels again at a later time.

To add a channel to your network, you will need to create a ConnectionConfig.

```
               globalConfig.ThreadAwakeTimeout = 10;

               ConnectionConfig connectionConfig = new ConnectionConfig ();


        }
```

This connection config will store all of your channels.

With the connection config setup to accept some channels, you can actually make your channels now.

```
               ConnectionConfig connectionConfig = new ConnectionConfig ();
               reliableChannelID = connectionConfig.AddChannel (QosType.ReliableSequenced);
               unreliableChannelID = connectionConfig.AddChannel (QosType.UnreliableSequenced);
```

Note the QoSType parameter passed into the AddChannel function. This is how you notate between a UDP/TCP connection. You start off with QoSType , and then further define what type of channel you want. There are a lot of channels to choose from, briefly outlined below:

- ReliableSequenced
  - TCP channel
  - Packets are guaranteed to arrive at the destination, even if they have to be resent. If packets do not arrive, the client is disconnected.
  - Packets are guaranteed to arrive in the order they were sent.
- ReliableFragmented
  - TCP channel
  - Packets are guaranteed to arrive at the destination, even if they have to be resent. If packets do not arrive, the client is disconnected.
  - Large packet sizes can be sent in multiple fragments.
- UnreliableSequenced
  - UDP channel
  - Packets are not guaranteed to arrive at the destination.
  - If a packet arrives out of order, it will be dropped.
- UnreliableFragmented
  - UDP channel
  - Packets are not guaranteed to arrive at the destination.
  - Large packet sizes can be sent in multiple fragments.

TOASTY TOES

After creating a channel configuration, you are ready to create a host topology. This is used to combine the channels with the number of maximum connections your server can have.

A very important thing about host topology, if the clients do not have the same host topology as the server, the client cannot connect. Of course, you can manually create a by-pass for this, but for the sake of your sanity you will not be doing that in this lab. For all purposes, if the client and the host do not have the same topology (meaning they have different channels mainly) then they cannot talk to each other!

```
               reliableChannelID = connectionConfig.AddChannel (QosType.ReliableSequenced);
               unreliableChannelID = connectionConfig.AddChannel (QosType.UnreliableSequenced);

               HostTopology hostTopology = new HostTopology (connectionConfig, maxConnections);
        }
```

INITIALIZE THE NETWORK?

With all of the busy-work complete, and all of your preparations at an end, you can finally initialize the network. In order to initialize the network, you will need to ask the Transport layer to prepare to open a socket. This is easily done with a single command: NetworkTransport.Init. You do need to tell the Transport layer what type of socket it is preparing to open. Luckily, you already defined this!

```
               HostTopology hostTopology = new HostTopology (connectionConfig, maxConnections);

               NetworkTransport.Init(globalConfig);
```

The next step is to actually open the socket on your network. When you open the socket, it will return to you a unique identifier that you can use to gain access to the open socket. To open a socket you need two pieces of information: the topology that the socket will use (which in turn is the channels and the number of connections the socket will support) and what port the socket will be punching through.

```
               NetworkTransport.Init(globalConfig);

               serverSocketID = NetworkTransport.AddHost (hostTopology, 7777);
```

Do not get confused about the “host” terminology. When you are working on the transport layer you are no longer working with the idea that a host is both a client and a server. Instead, the host refers to the socket information!

At this point, you can add a few debug statements to determine if the socket was successfully opened. Remember that you initially assigned the serverSocketID to -1. If you were able to successfully open a socket, then your serverSocketID will not contain that unique identifier. If you failed to opened the socket, the serverSocketID will still be -1!

```
        serverSocketID = NetworkTransport.AddHost (hostTopology, 7777);

        if (serverSocketID < 0) {
               Debug.Log ("Server socket creation failed!");
        } else
               Debug.Log ("Server socket creation success");
```

Last, set the variable that represents if the server is running to true (assuming that the socket was successfully created).

```
    // Use this for initialization
    void Start () {

           GlobalConfig globalConfig = new GlobalConfig ();
           globalConfig.ReactorModel = ReactorModel.FixRateReactor;
           globalConfig.ThreadAwakeTimeout = 10;

           ConnectionConfig connectionConfig = new ConnectionConfig ();
           reliableChannelID = connectionConfig.AddChannel (QosType.ReliableSequenced);
           unreliableChannelID = connectionConfig.AddChannel (QosType.UnreliableSequenced);

           HostTopology hostTopology = new HostTopology (connectionConfig, maxConnections);

           NetworkTransport.Init(globalConfig);

           serverSocketID = NetworkTransport.AddHost (hostTopology, 7777);

           if (serverSocketID < 0) {
                  Debug.Log ("Server socket creation failed!");
           } else
                  Debug.Log ("Server socket creation success");

           serverInitialized = true;
    }
```

CHECKPOINT!

To verify that this is working:

1. Assign the ServerConnection script to the Server object in the hierarchy.
2. Run the game.
3. Your console should say “Server socket creation success”

STEP THREE: RESPONDING TO THE NETWORK

The next step is setting up the server to respond to clients. Every frame you will want your server to check and see if any messages have come across the network. However, you only want to check for messages if the server is running on a socket!

```
    // Update is called once per frame
    void Update () {
           if(!serverInitialized){
                  return;
           }
    }
```

At this point in time, there are only a few messages that can be sent across the network. These messages are:

- Nothing
- Connection
- Disconnection
- Data

The “Nothing” message is basically says that no new messages have arrived.

The Connection message is sent when a client successfully connects to the server.

The Disconnection message is sent when a client disconnects from the server.

The Data message is sent when a client chooses to send data to the server.

All of these messages work the other way as well (so the client receives a connection message when they successfully connect to a server, a disconnection message when they disconnect from a server, and a data message when the server chooses to send data to the client).

There is a little bit of bookkeeping involved in sending a receiving these events. You need to keep tract of who is sending the message, who is receiving the message, what channel the message is being sent or arriving from, how large the message can be, what the message is, and finally if there was an error.

```
        if(!serverInitialized){
               return;
        }

        int recHostId;              //who receiving the message
        int connectionId;     //who sent the message
        int channelId;        //What channel the message was sent from
        int dataSize;         //how large the message can be
        byte[] buffer = new byte[1024];     //the actual message
        byte error;                 //if ther is an error
```

And finally, you need to know what type of message was sent! (Connection, disconnect, or data).

```
        byte error;                 //if ther is an error

        NetworkEventType networkEvent = NetworkEventType.DataEvent;
```

Now that you have all of your bookkeeping taken care of, you are prepared to receive a message.

Since messages are sent in a queue, you will want to make sure that you are going through the queue everytime you are checking for messages (you want to process as many messages as possible!). You can accomplish this with a do-while statement (do process all messages in the queue while there are messages in the queue).

```
        NetworkEventType networkEvent = NetworkEventType.DataEvent;

        do
        {
```

```
		} while ( networkEvent != NetworkEventType.Nothing );
```

The first thing you want to do is receive the message from the Transport layer. You do this in a single statement:

```
		do
		{
			networkEvent = NetworkTransport.Receive( out recHostId , out connectionId , out channelId , buffer ,
1024 , out dataSize , out error );

		} while ( networkEvent != NetworkEventType.Nothing );
```

NetworkTransport.Receive(out int receivingID, out int sendingID, out int channelID, byte[] bufferToStoreMessage, int bufferSize, out int sizeOfMessage, out byte error)

After receiving the network event, it comes down to doing a simple switch statement depending on what type of event you have receiving! As stated earlier, there are currently only three types of events: Connect, Data, Disconnect (and nothing, but that doesn't really count).

```
		do
		{
			networkEvent = NetworkTransport.Receive( out recHostId , out connectionId , out channelId , buffer ,
1024 , out dataSize , out error );

			switch(networkEvent)
			{
				case NetworkEventType.Nothing:
					break;
				case NetworkEventType.ConnectEvent:
					break;
				case NetworkEventType.DataEvent:
					break;

				case NetworkEventType.DisconnectEvent:
					break;
			}

		} while ( networkEvent != NetworkEventType.Nothing );
```

PROCESSING EVENTS

Most of the event processing is rather simple. You can do almost anything you want in these events, such as spawning players, setting up variables, changing the scenes of the player, or similar tasks that the server might need to do when a client connects or disconnects.

Starting with the simplest event: Nothing. If you receive an event of Nothing, then you really don't want to do anything at all. What this means is there are no messages left in the queue, there is nothing to do! If you receive an event of Nothing, then you just want to continue on.

The connection and disconnect events are just as simple for this lab, you just want to output that a client has connected or disconnected **from the server.** This is important, as you want the server to always know when people are connecting to or leaving your game!

In order to verify that the client is connecting **to the server**, you can check the serverSocketID that you stored earlier when you opened the server socket, to the receivingID that was sent with the message (remember: not all messages reach their intended targets – through routing and various other factors your network can possible get traffic that you aren't expecting. You always want to make sure that you are only processing messages that you are expecting to receive!)

After verifying that the server was the intended target of the message, you can display a simple debug statement displaying the ID of the connection that was created between the server and another computer.

You follow the same procedure for the disconnecting event: Verify that the server is the intended target, and display a debug displaying the ID of the connection that disconnected from the server. In more advanced games you would do other things in this event such as destroying player objects and other tasks.

```
		do
		{
			networkEvent = NetworkTransport.Receive( out recHostId , out connectionId , out channelId , buffer ,
1024 , out dataSize , out error );

			switch(networkEvent)
			{
				case NetworkEventType.Nothing:
					break;
				case NetworkEventType.ConnectEvent:
					// Server received disconnect event
					if( recHostId == serverSocketID ){
						Debug.Log ("Server: Player " + connectionId.ToString () + " connected!" );
					}
					break;
				case NetworkEventType.DataEvent:
					break;
				case NetworkEventType.DisconnectEvent:
					// Server received disconnect event
					if( recHostId == serverSocketID ){
						Debug.Log ("Server: Received disconnect from " + connectionId.ToString () );
					}
					break;
			}

		} while ( networkEvent != NetworkEventType.Nothing );
```

RECEIVING DATA

The last type of message you can get is a data message – this is a client sending some type of data over the network. The easiest type of data to work with is a string.

Sending and receiving actual data has it's own complications: remember that you are running at this point in the transport layer of the protocol. This means that you will need to manually encapsulate the data that you want to send across the network.

On top of that, you also need to make sure that the data is serialized so the unity application knows what to do with it!

The first thing that you need to do is setup some sort of stream reader. Normally in C# you would use a StreamReader for this. However, in this case you are going to use a very specialized version of a stream: a MemoryStream. A memory stream is a stream of data (like a stream reader, or stream writer, or just a basic stream) that does not access the hard drive or disk space on the computer – it functions 100% in the memory of the computer. Data that is stored in the memory of a computer is much faster to access than data on a disk. However, because this information is stored in memory there is no file or other resource that contains this data. One of the catches of a memory stream is that they typically work in binary – the base "language" that computers (and memory) works off of.

In order to actually read something from a MemoryStream, you need to be able to read the binary information that the MemoryStream will provide. Of course, you could simply use a BinaryReader to accomplish this task for you. However, keep in mind that you still have to deal with the fact that this information is serialized on top of everything else! To make life easier, there is a class that you can use that will both deal with the binary information AND work with the serialization of the information. This is called a BinaryFormatter.

You can use the BinaryFormatter to process the information from the MemoryStream, Deserialize the information (remember: serialized information is sent over the network and must be deserialized to be accessed, this is like the encapsulation of the Transport layer. It is encapsulated on one computer, sent across

the network, and need to be un-encapsulated at the target computer for the Transport layer to read the information), and finally turned into a string that humans can read with ease!

```
case NetworkEventType.DataEvent:
       if( recHostId == serverSocketID ){  //verify the server is the intended target

              //Open a memory stream with a size equal to the buffer we set up earlier
              Stream memoryStream = new MemoryStream(buffer);

              //Create a binary formatter to begin reading the information from the memory stream
              BinaryFormatter binaryFormatter = new BinaryFormatter();

              //utilize the binary formatter to deserialize the binary information stored in the memory string
              //then convert that into a string
              string message = binaryFormatter.Deserialize( memoryStream ).ToString ();

              //debug out the message you worked so hard to figure out!
              Debug.Log ("Server: Received Data from " + connectionId.ToString () + "! Message: " + message );
       }
       break;
```

Whew! Now that was a lot of work.

SENDING A MESSAGE

Now that you can receive a message, it is time to piece together how to send a message. Luckily, this works almost exactly the same way!

The most important thing is to make sure the message you are sending does not exceed the buffer size (in this case 1024).

For sending a message, you will need:

- A buffer to store the information
- A MemoryStream to send the information
- A BinaryFormatter to serialize and convert the message into binary
- A way to store if any errors arrive

```
void SendMessage(string message, int target)
{
       byte error;
       byte[] buffer = new byte[1024];
       Stream memoryStream = new MemoryStream(buffer);
       BinaryFormatter binaryFormatter = new BinaryFormatter();
}
```

With the bookkeeping taken care of, alls that is left is to serialize the message and send it on its way!
Just like the BinaryFormatter could deserialize binary from a stream, it can also serialize binary into a stream. This is done by utilizing BinaryFormatter.Serialize(Stream stream, string message).

```
void SendMessage(string message, int target)
{
       byte error;
       byte[] buffer = new byte[1024];
       Stream memoryStream = new MemoryStream(buffer);
       BinaryFormatter binaryFormatter = new BinaryFormatter();

       binaryFormatter.Serialize (memoryStream, message);
}
```

And lastly (almost) you can send this message down through the rest of the networking layers to the target recipient! In order to send the message from the Transport layer you need to specify who specifically is sending the message, where the message is going, what channel you are sending the message over, what information you are sending, the size of that information, and if there is an error.

```
binaryFormatter.Serialize (memoryStream, message);

//                Who is sending,     where to, what channel,   what info,    how much info,      if there is an error
NetworkTransport.Send (serverSocketID, target, reliableChannelID, buffer, (int)memoryStream.Position, out error);
```

Note that memoryStream.Position is similar to Stream.Position, it just states where in the stream the reader/writer is at (it starts at 0, as it writes into the stream it increases for every character written).

Last but not least, you need to catch any errors that might have occurring during the sending process.

You can use the NetworkError enum to determine if there was an error, and if so print out some very basic information about the error. This is the same method that you would use for basic exception handling in most programs.

```
NetworkTransport.Send (serverSocketID, target, reliableChannelID, buffer, (int)memoryStream.Position, out error);

if( error != (byte)NetworkError.Ok) //error is always assigned, and it uses Ok to notate that there is nothing wrong
{
       NetworkError networkError = (NetworkError) error;
       Debug.Log ("Error: " + networkError.ToString ());
}
```

ACTION :: REACTION

Finally, you received your message. Now it is time to do something with it!

You will do a very simple transition – if the client sends the server a message telling the server that it is the clients first connection, then the server will order the client to move to a different scene. If the server is not already in that scene, then the server will move to that scene as well.

Create a new function named RespondMessage, and take a string (for the message) and an int (for the sending client's unique identifier) as a parameter.

Inside of the function, check to see if the message was "FirstConnect". If it was, debug out to the console that you received a first connect message from the player, and print out the player unique ID.

Then, send a message back to that player telling the player "goto_NewScene".

Finally, check if the server is currently in a scene named "Scene2". If the server is not, tell the server to load up the scene.

Once you complete that, add a call to the function in the NetworkEventType.DataEvent.

```
case NetworkEventType.DataEvent:
       if( recHostId == serverSocketID ){  //verify the server is the intended target
```

```
        //Open a memory stream with a size equal to the buffer we set up earlier
        Stream memoryStream = new MemoryStream(buffer);

        //Create a binary formatter to begin reading the information from the memory stream
        BinaryFormatter binaryFormatter = new BinaryFormatter();

        //utilitize the binary formatter to deserialize the binary information stored in the memory string
        //then conver that into a string
        string message = binaryFormatter.Deserialize( memoryStream ).ToString ();

        //debug out the message you worked so hard to figure out!
        Debug.Log ("Server: Received Data from " + connectionId.ToString () + "! Message: " + message );

        RespondMessage(message, //who send the message so the server can respond to that specific person);

}
```

```
void RespondMessage(string message, int playerID)
{
        //Student will fill this out
}
```

Lastly, add the DoNotDestroyOnLoad(this); statement in the start function so this script will persist between the scenes.

STEV FOUR: NEW SCENE

Create a new scene named “Scene2”. Add a cube to the scene at position 0,0,0. Add this scene to the build settings. Return to the original scene (the one with the Player and Client objects).

WRAP UP

You now have the ability to open a socket and send a binary message over the network. Although this is a starting point to networking in lower-level APIs and lower on the networking models, it is invaluable knowledge that you can apply to any basic network and expand upon.

THINK ABOUT IT!

1. What does the connectionID mean when the client receives messages from a server

ON YOUR OWN!

1. Finish the RespondMessage function on the server connection
2. Complete the ClientConnection. The template for this script can be found on D2L as well on the Github.

   You can verify that this is working by:
   1. Put the ClientConnections.cs script on the Client game object
   2. Put the ServerConnections.cs script on the Server game object
   3. Disable the ServerConnections game object (server)
   4. In unity, go to Edit > Project Settings > Player and check mark “Run in Background” under Resolution and Presentation > Resolution.
   5. Build the game
   6. Enable the ServerConnections game object (server)
   7. Disable the ClientConnections game object (client)
   8. Run the game in UNITY and verify that the server socket is opened via the debug
   9. Run the build game and verify that the client connected via the debug in UNITY
   10. In the built game, press the spacebar and verify that both the client and server moved to the new scene with the cube in it (“Scene2”)
   11. In the built game, press the R and verify that a message was sent to the server via the debug in UNITY
   12. Close the built game and verify that a message was sent to the server via the debug in UNITY