**Unity Multiplayer Server/Client**

Server Client code for this project is based off a C# library known as ***Lidgren***

* Establish connections, and send messages over the internet, and over the network to other machines
* Dave’s version is customized to work with Unity
  + Dave’s networking code is like Photon/Unity’s where there are rooms and lobbies.
  + Using Unity’s Photon makes you pay $100 a month for 100 connections at once.
  + Using your own server, and making your own network code means no monthly fees

Layered on top of Lidgren is the networking library customized for Unity (UCNetwork)

* UCNetwork allows you to have *authoritative DNS servers* (unlike Photon/Unity)
  + *authoritative DNS servers:* stores the maps of domain names to IP addresses
  + Photon/Unity usually peer-to-peer with master-client relationships
* All the clients don’t need to be on the same client version to connect to UCNetwork
* Many different types of clients (cellphones, WebGL, PS4 VR, Xbox, and windows clients)

**ExampleServer GameObject**

* Example Server script
* Server Network script
* Console – feed for information
  + If you make a build so there’s an executable (-headless mode) flag
    - Loads it without a UI
* Hit play on Server, it does nothing but load up the server

EDIT -> Project Settings -> Player (need to make sure Run in Background is checked)

UCNetwork class script

* Core Networking code
* Handles data transmission under the hood

Server Network class script

* Specific code for hosting a game server
* Has the maximum number of connections that you want
* Should it be writing out to a log flag in editor

**Example Server class**

* Uses the Server Network class to create a new server
* Server Network object
* Server Name, port number
* What it does:
  + First – its setting up the port of the server network to be equal to the port number listed in inspector
  + If no server network component, it tries to get a server network component from the gameObject.
  + If the Example Server class can’t find a server network component, then it adds a server network component
  + Creates the server network, and that is pretty much it
* Whenever a new client connects to the server, you will get a connection request.
  + As a response to this connection request, whether you want to approve the request or deny the requests (over the maximum allowed connections).
* You also get a callback whenever a client has connected to your game
* You get a callback when network object is being created
* Callback when a client has been disconnected
* Callback when network object is being instantiated.
* Callback for areas

SyncUpdate

* The updates going over the network from that player object instantiated by the Example Client GameObject’s Client Network script basically saying, I’m changing position, everyone else should be doing the same thing on your local clients for this networked object.

Networked Object

* An object that is synchronized over the network
* For example:
  + If you have multiple players in the same area, the walls and the floors are ***NOT*** networked objects. (not moving, static scene object, loaded on local client)
  + My player, and all other players where the position changes needs to be kept in-sync. These are networked objects
  + Position updates, functions back and forth, animations that need to be synced up are typically stored as networked objects
* Every Networked Object has it’s own UniqueID

**ExampleClient GameObject**

* Our client is going to have all the typical items, game code, Prefabs, scene setup
* In this example:
  + Canvas Object
    - Asks for Address and a Port, and a connect button
  + TitleScreenLogic Object
    - Connect button
      * Calls TitleScreenLogic script.Connect function
      * Gets example client object
      * Calls ConnectToServer
        + Server.text, int.Parse(port.text));
  + To Connect to a machine, you need 2 things: IP and Port
  + Port tells the machine which application is listening for it
  + My machine might be running many applications using the network
  + when incoming data is coming in, it needs to know what application to send that to
  + Port forwarding not required on local machine or local network

**Example Client Script**

* Client Network Object: Inherits from Client Network with specific code tailored for your game
* If not client network object on the example client script, add it to script
* Connect to Server function (string server Address, int aPort) when button is clicked
  + To connect server, we need to specify the port to use,
  + ClientNet.connect(server name, port, username, password, clientType, uniqueID)
  + On the server when it gets this connection request:
    - It gets the data just sent in from this function call
      * Data is a struct with all the above listed data
      * Server then approves Connection by (data.id)
  + Here’s 3 strings and integer number that you can send to the server as your connecting just to tell the server who you are, and server then does its connect
* As the client connects, or disconnect, there are a bunch of different callbacks that print messages to the debug log
* OnNetStatusConnected()
  + ClientNet.AddtoArea() – calls clientNet class to addToArea
* Public OnChangeArea()
  + Called from ClientNet
  + Instantiate player (ClientNet.Instantiate – instantiates a networked object for all machines)
  + Get NetworkSync component script
  + Get Player component scripts

**ClientNetwork**

Client Network on its own doesn’t have any specific logic for connecting to a server, or specific gameplay for our game, or instantiating our game.

The Client Network is for connecting to a server, and sending data back and forth primarily.

* Am I connected?
* Do I have some UniqueIDs I can use?
* Creates an Object locally
* Creates a NetOutgoingMessage
  + Goes up to the server
  + Sends a network message to the server
  + Server receives this message, and as a response, it going to track that there is a networked object there, and then it’s going to send out a message to all the players information about this networked object.

ClientNetwork assumes almost all objects that you are instantiating over the network has a network sync script attached to it.

You can create objects over the network that don’t have network syncs on them. (aka instantiate via RPC).

***Network Instantiate needs 3 things***

* CreateLocalObject (see below)
* NetOutgoingMessage = CreateMessage function( Lidgren message packet )
* AddOwnedID (newID) function – adds this item to our list of network ID’d objects
  + newID is returned from the CreateLocalObject function

***Create Local Object needs 3 things***

* Unity’s normal Instantiate function (takes a prefab stored in Resources folder)
* NetworkSync script (not required, but 95% of the time will need to have one on prefab)
* Network Initialize using SendMessage() function

**Connection** – when a client goes to connect to a server (57:55, 11-14 video)

* ExampleServer.ConnectionRequest();

**Ownership**

* This networking library has the concept of ownership
* Objects you own are any objects that your client instantiates
* Owning an object means you are the authority on what this object is doing
* When you disconnect from an area where you are the owner of several objects, and another player exists and, then the other player (or some other player in the same area) must gain ownership of those objects.
* Server automatically transfers ownership to players as an optimization
* As you gain or lose ownership, the Network Sync will send callbacks

“If this is a Server-Client authority network library, isn’t the server making the authoritative decisions already? Why does a particular player need to take ownership of an AI enemy or something, when the server is going to have the final say for what happens to the monster?” -Ryan

* + The only thing that ownership does is it allows your client to send position updates to the server.
  + This gives you performance optimizations for more complex objects that use physics.

**SendMessage *() –*** *specific for this library (there is another Unity SendMessage function)*

* SendMessage allows you to call functions inside of other scripts without calling function directly. It looks for all the scripts attached to the object that calls this SendMessage function, and if any of the scripts have a function that is specified in the SendMessage parameter, that function will be called. (Same as RPC). This is how all Callbacks work.
  + GameObject myGameObject;
  + myGameObject.SendMessage (“MyTestFunction”, SendMessageOptions. )
* In the example above, the SendMessage function will look at myGameObject, and find all the scripts attached to myGameObject. It will search each script for a function called, “MyTestFunction”. If that function is found, it will call that function.
* Doesn’t require a Network Sync. Won’t break if no functions are found.

**NetOutgoingMessage sendMsg = client.CreateMessage();**

For sending messages back and forth to the server and clients

* client is a NetClient object which is part of the Lidgren library
  + First, we create a sendMessage object
  + Second, we write all the data that we want to put in that message
  + Third, we call client.SendMessage function which sends the message.
    - client.SendMessage (msg, deliveryMethod)
  + Last, we call AddOwnedID
    - Adds ID to list of owned IDs.

**HandleMessage\_Data** (on both the clientServer and ServerNetwork scripts)

* We just got a message we got over the network, we should do something with this data

***Server:***

* The first thing the server does is reads the first integer from the message
  + An integer which uses an Enum of MessageType (1-14)
  + First, I need to figure out what type of Message this is, so I can read rest of the message data
* Second thing is grab the unique Client ID for who sent this to me (grab from server)
* Next, a series of if statements
  + Is Message.RPC?
  + Is Message.SyncUpdate || Message.LiteSyncUpdate
  + Is Message.Instantiate message
    - Get message data from instantiate message
      * Read all the values from the instantiate message
      * Log it
      * Set up position and rotation
      * Subtract number of IDs (if less than half, give more IDs)
    - InstantiateNetworkObject
      * Server keeps track of that networked objects, and tells other clients to instantiate this object
* Server keeps track of all the objects you own (Line 1074)
  + Foreach(ClientData data in clientData)
    - Data.ownedObjects.Add(aNetworkId)
  + If spawned by the server (spawns object and transfers ownership)
* If you need to know data about any of the objects connected to the server:
  + Line 1120 (inside networkObjects map based on Unique network ID)
* SendMessages – let example server object or any other script on this script, let them know when this instantiate happened.

**HandleMessage\_Data** (on both the clientServer and ServerNetwork scripts)

* We just got a message we got over the network, we should do something with this data

***Client:***

**PlayerPrefab / NetworkSync**

Client Network assumes almost all objects that you are instantiating over the network has a network sync script attached to it.

You can create objects over the network that don’t have network syncs on them.

Whenever a client connects to the server they are given 500 IDs by default.

* Whenever you want to instantiate a network object, you (being the client) instantiate that object locally and give it one of your unique Network IDs. Then tell the server that you’ve instantiated this object with this unique network ID. The server tells everyone else connected to the server to instantiate that networked object locally on their machine with that network ID.

Network Sync gives a unique aNetworkID for each object in-sync on the server

Server has its own set of Unique IDs for the clients that connect. (Not Applicable for Net Sync)

When we talk about UniqueIDs for NetSync, I’m referring the unique ID assigned to every object that is connected to the server with a Network Sync script

* NetworkSync Script on player keeps gameObject in-sync over the network.

**Broadcast Frequency**

* Not a guarantee it will go smoother (network traffic increases)
* Sends a message to the server, server processes it, then sends it back down to all the other clients
* Broadcast smarter with L.E.R.P.
* Broadcasting at 10 times a second doesn’t mean we are receiving a message 10 times a second.
* We need to account for that sometimes a network message is going to be slower (networking hardware, network infrastructure)

**Delivery Method**

* Reliable ordered
  + Nope, every message over the network, needs to get there
  + If a message gets dropped, be sure to have client to re-send the message
  + If a message that gets there that’s out of order, delay processing that message until the message that was supposed to be there before that gets there first.
* Unreliable Sequenced
  + If I send you two position updates, and one of them gets dropped, what happens is the server will figure out one of them got dropped (no confirmation), and the client will send it again. The first position update is behind it.
  + If this message gets dropped, don’t worry about it. Don’t send it.
  + If these ever get swapped, dropped the ones that are out of order

**RPC Notes: (Remote Procedure Call)**

**Send data over the network without Network Sync (uses networkIDs instead)**

Preface about **<networkID>**

Whenever a networked synced object is instantiated, that object gets a unique **<networkID>**

The Network Sync script gives a unique **<networkID>** for each object in-sync on the server

All network synced **<networkID>** objects are given ownership to a client connected to the server

Each client has a list variable containing 500 available network IDs when they connect to the server.

Whenever a client connects to a server, that client’s networked owned objects will each get their own **<networkID>**

When a client gains ownership of a networked object, all networked objects owned by that specific client get added to their **<networkID>** list.

The server has its own list of unique IDs for each client connected to the server, BUT… that is NOT the **<networkID>** in clientNet.CallRPC’s function param list.

**What’s a Remote Procedure Call**?

* Asks the Server to Tell all other clients in the server to run this function

**clientNet.CallRPC(“FunctionName”, UCNetwork.MessageReciever.AllClients, < networkID>, params[])**

// This sends a message to all clients to call “FunctionName” on the object with **< networkID>**, and passing in optional params[] as part of the this RPC Call.

// IN THE CASE of clientNet.CallRPC function, the **< networkID>** parameter is asking for which networked object should this RPC call go t,

// THE RPC then looks for that function name on the object with <networkID> and calls it if a script is attached to that <networkID> object which contains the function by the same name.

* Essentially by including <networkID> not -1, the client wants a specific gameObject (that has a network sync) to receive a specific RPC function call. That specific object will be updated with the function’s logic and applied to all clients connected to the server
* If the **<networkID> param is -1**, the GameObject running the **Client Network script** will look for the function name passed in (or any other scripts attached to same GameObject
* Params[] – parameters passed in to RPC Call should match signature of defintion

FOR EXAMPLE:

**GameObject Monster is instantiated with a Network Sync and given <networkID> of 102**

**clientNet.CallRPC(“DamagedMonster”, UCNetwork.MessageReciever.AllClients, 102, 4.9)**

* “I want this monster with <networkID> 102 that I just dealt 4.9 damage to have an RPC called on itself on all the other clients machines.
* This damage update will be reflected on all clients connected to the server and update that specific network-sync’d monster’s stats.
* Because I own that monster, I know the monsters <networkID>.
* Because a function named void DamageMonster(int dam) was present on a script attached to that <networkID>’d object, that function was able to be called on that specific monster, and the stat update was also reflected on all clients also connected to the server.

**Other:**

*Abstract function* in parent class – To use that function, the child class must implement a definition for that abstract function. In addition, the parameter definitions need to match up exactly. So, whatever parameters you plan to need in the child class’s version of the abstract function, be sure to include those parameters in the Parent’s Abstract function prototype.

*MonoBehavior-* Is the base class for every script component in Unity. If you want to add a script to a gameObject as a component, that script will inherit from MonoBehavior.

*Unity Special Folders (Editor, Plugins, Resources)*

* Plugins – if you put a .dll in your Plugins, Unity will automatically add it to your solution.
* Resources – all prefabs you want to instantiate over the network needs to be put in this.

Lesson Learned on putting right components in right places for network Prefabs:

\* Calling RPC from player input script

\* I want the Remote\_Anim\_Calls script to recieve RPC

\* I want all the clients to play back my local animation on

their machine

\* they need access to my booleans -or- to set their own booleans in relation to me.

Dave: That's just the first part (about putting the animator componenent in the prefab).

The second part is the remote anim script. Because in order for your player client to see my animation, I need to send information to that script saying I'm playing my animation now.

\* How do I store the booleans for all the players connected?

\* How does the client receive the proper network ID for the player associated with the animation rpc call?

\* How does the other player associate that with booleans stored on that remote script?