Lidgren Basic Setup API

This document will take you through the basic API and code pieces of the “Lidgren Basic Setup” package from the Unity Asset store.

First of all lets discuss the parts of the code which are pretty standard and do not contain any networking or Lidgren specific pieces. If you look in the LidgrenExample/Scripts folder, you will see five files: PlayerAnimator.cs, PlayerController.cs, RPGControllerUtils.cs, RPGThirdPersonCamera.cs and SelectUI.cs

Let's give them a quick overview, one by one:

- **PlayerAnimator –** Controls the animation of the player characters, most of the code in here is standard Unity code and nothing special. But we should look into one method in more detail: OnPlayerMovement(byte newState). This method gets called whenever we want to switch animations, but the top part of it is a bit special. It checks if we're the current owner of the object and if we are, sends the animation change to the server.
- **PlayerController –** This is a simple character controller that uses rigidodies to move the player around the world. The only special part in this class is the Start() method which checks if we're the current owner of the object or not, and if we're not – it removes the controller from the object (so we can only control our own character).
- **RPGControllerUtils –** Class used by RPGThirdPersonCamera for reading inputs, nothing special here.
- **RPGThirdPersonCamera –** The camera class, a very simple third person camera which is locked behind.
- **SelectUI –** A simple class which is used in the “Menu” scene to render two buttons and then load the “Level” scene after Server or Client has been selected.

The next files I want to look at is the meat of the networking code, they are located in LidgrenExample/Scripts/Lidgren. I will not look at them in alphabetical order, but rather a logical order.

- **LidgrenGameObject –** This is a relatively simple script, it's attached to the Player prefab (found in LidgreExample/Resources). The first two things to look at is the Id and IsMine fields on the script. They are set internally by one of the other classes. Id is the number that identifies a client across the network, IsMine is set when the object is spawned to either true or false, if the Id supplied for the spawned object matches our local clients id. The Connection property is also set internally when the object is created, we will look into this later.

  The next thing to look at is the FixedUpdate function. It checks if “IsMine” is true, and if it is – it sends a message with both the position and the rotation of our game object to the server.

  The Spawn(int myId, int id, NetConnection con) method Is used to create a new instance of the Player prefab and set up it's LidgrenGameObject.

- **LidgrenMessageHeaders –** A very simple, static class – it's just a utility that holds the different bytes we use as message headers to identify the packets we send across the wire. If you looked closely at the FixedUpdate function of the LidgrenGameObject script you will see that we use it there to signify that the position package we send is of type of “LidgrenMessageHeaders.Position”

- **LidgrenPlayer –** This is a very simple class that is only used on the server. It contains the Id of the player and the LidgrenGameObject that this player owns.

- **LidgrenPeer –** This is the first class we've come across with a lot of networking specific code in it. Let's step through it with care.

The first thing you will see is the two private fields “peer” and “messageHandlers”. The peer field will contain an object of the type NetPeer which is a class internal to Lidgren that represent a client or server. The field messageHandlers is a dictionary that that contains a number mapped to a delegate that takes an incoming message as it's only parameter. This is used for handling messages (or packets) that arrives.

Next you will see two events, Connected and Disconnected. Pretty obviously they are called when the client connects or disconnects to/from the server, or when the server receives/closes a new/old connection from a client.

SetPeer and RegisterMessageHandler are two utility functions, used to set the peer object and add a new message handler to the messageHandlers dictionary.

Now, time for the bulk of the code in this class, the ReadMessages method. This method will, if peer is not null. Read a message, and if it successfully reads one (not null) it will check it's type. When the type is data it will read the first byte (remember the LidgrenMessageHeaders class? The first byte we read out here map to a value in there) and then try to get a delegate out of the messageHandlers dictionary that has that number as a key. If it finds a delegate it will be invoked with the message as it's only parameter. This is how we map network messages sent with Lidgren to actual functions being called in code.

It will also check for the special message type “StatusChanged”, which means we either connected or disconneced, and if we did we invoke the proper event.

And even further down it will check for the standard messages for debugging and error reporting, and display them using the unity Debug.Log function.

Last but not least, we will recycle the message (this allows Lidgren to re-use the message for future messages, saving us memory and performance).

We also call ReadMessages() in all three update functions supplied to us by the Unity engine.

- **LidgrenServer –** This class extends the LidgrenPeer class, it has one field you can modify from the inspector – what port it should run on. Let's step through it also.

  The first thing we should look at is the Start() method. First it will make sure that the server is not destroyed when we load a new scene. This is what allows us to start the server in the “Menu” scene and still have it running after we loaded the “Level” scene.

  We then create a very basic NetPeerConfiguration object, which is the lidgren configuration. Set it's port and then create a Server object by supplying the config object to it and then starting it. We then call the SetPeer() method that we inherited from LidgrenPeer to set the NetServer object as the peer.

  We then subscribe to the Connected and Disconnected event so we know when clients join/leave us. And last, we map three of our internal functions to three of the byte values from LidgrenMessageHeaders so that these functions will be called when we receive a message that begins with this byte.

  Let's investigate the internal methods in this class. The first one we find is spawnOn, which is a utility method that takes a LidgrenGameObject and a NetConnection, it creates a message and writes the spawn header byte and then the id of the game object and sends it to the connection. This is what allows the server to spawn objects on the clients.

  The next function “onMovement” is one of the functions we registered in the Start() method, and it will be called (from inside LidgrenPeer.ReadMessages) whenever we receive a message where the first byte is the same value as LidgrenMessageHeaders.Movement. Basically what we do here is that we extract the LidgrenPlayer object from the connection (they are connected together further down in this file), we then create a new message, copy the entire contents of the message we received and send it to all client,

except the one that sent the message. This is the piece of code that distributes the new movement events across the network for us. We then read the message ourselves (and throw away the Id, as we don't need it, the server already knows which client sent the message) and call into the PlayerAnimator.OnPlayerMovement method to update the animation.

Next up is “onPosition”, it works exactly like onMovement – it copies the entire message (that contains our position and rotation) to a new out going message and sends it all clients (except the one that sent it to the server) – it also reads it on the server so that the server gets the new position also.

Our next function is called onRequestSpawn, it's the last one that was mapped to a byte in the Start() function. When a client is ready to have it's character spawned, it will send a very simple message consisting of one byte – the “LidgrenMessageHeader.RequestSpawn” byte – to the server, this will cause the server to spawn the object (if it doesn't already exist, we only want each client to have one character!) on the server, and also spawn it on all other clients by calling spawnOn on the new object + every client connection.

Next up is our two event handlers, onConnected and onDisconnected. They do pretty much the reverse of each other. When a client connects, onConnected is called. We create a new message with the header byte “LidgrenMessageHeaders.Hello” and write the client id to it also – and send this to the new connection.

We then create a new LidgrenPlayer object, supply the same client id to it and attach it to the NetConnection objects .Tag (this it the .Tag property that we use in onPosition, onMovement and onRequestSpawn to get the player object from a connection). We then send our hello message to the client with it's id in it.

After that we find all existing LidgrenGameObject instances, and spawn them all on the new client by calling spawnOn with the game object(s) and the connection.

The onDisconnected method does the reverse, when a client disconnects we will extract it's LidgrenPlayer from the .Tag property and then send a “LidgrenMessageHeaders.Despawn” message to all other clients. We also destroy any LidgrenGameObject for attached to this player.

- **LidgrenClient –** The last file, LidgrenClient is very similar to LidgrenServer but obviously deals with the client instead of the server. First, there are two fields you can configure in the inspector – the IP and Port we should connect to. The Start() method is very similar to the server, except for a few differences: We don't set the port on the NetPeerConfiguration object. We also call both .Start and .Connect on the NetClient object we created. We also attach a couple of more message handlers to byte values by calling RegisterMessageHandler.

  The two first functions, onConnected and onDisconnected are very simple – we just log to the unity console that we connected/disconnected.

  Next up is onMovement, it works almost exactly like the onMovement we found on the server. But let's step through it. We first read out an integer from the message, this is the player id of the player that the message originated from. We use this to find the correct players LidgrenGameObject in our “lgos” dictionary. If we find one, we grab the PlayerAnimator component on it and call OnPlayerMomvement to update the animation state.

  A few lines down you will see onPosition, it works exactly like onMovement – but will instead update the position and rotation of our LidgrenGameObject instead of the animation state.

  Now, two interesting functions are up onSpawn and onDespawn. The first one gets called when we receive a “Spawn” message header byte from the server, this causes us to create a new LidgrenGameObject locally and spawn a Player prefab using the LidgrenGameObject.Spawn function. We also read out the first integer from the message to get the player id this spawn is for, so we know inside the .Spawn if it's “our” character or someone elses.

The reverse of onSpawn is onDespawn, very simple – we just read the player id and try to destroy the LidgrenGameObject associated with it and then remove it. If we fail it doesn't matter so we just catch all exceptions and ignore them.

The last function is onHello, which is sent form the server to a newly connected client, we read out the clientId assigned to us, and then we send a response back to the server, asking it to spawn our character.

That was every single file of the library gone through in great detail. For questions I direct you to the thread on the “Assets and Asset Store” forum on the official Unity Forums.