Network Communication Model

I use two different types of network communication.

1. RPCs and Commands from the player object (prefab)
2. Network Messages

RPC's and Commands from a prefab

These are best when you want to send a message out to all clients, including your local client (if using local host). This method is really easy to use and can be used in most cases. Within the RPC you can also exclude the server. This makes it really easy to be used for all clients or all clients that are not the server.

Using this method you must have the NetworkIdentity component on the prefab. I have Local Player Authority set to true. The script with your Commands and RPC's must also be on this prefab. Command methods must start with Cmd. RPC methods must start with Rpc.

Here is an example where I'm setting up some dice for a dice roll in my game.

These methods are right beside each other on the prefab script. This makes it easy to follow the network links between server and client. The command is called from the server and passes along die information. The RPC is called within the command. The RPC will run on the local client if you have one so if you don't need it to function you can check if the client is also the server and exit. Otherwise I have a singleton dice manager that takes the die information and uses it on all the clients.

You can only pass certain types with an RPC. The list is at the bottom of this page: http://docs.unity3d.com/Manual/UNetActions.html

The DieFace array is my own struct that contains allowable types (string, int, etc)

```
 [Command]
public void CmdAddDieToClients(string diceBlueprint, DieFace[] dieFaces ){

              RpcAddDieToClients(diceBlueprint,dieFaces);
}

[ClientRpc]
public void RpcAddDieToClients(string diceBlueprint,DieFace[] dieFaces){

       //The server has already performed this action and we do not need its local client
       //to perform the action again.

       if(isServer)
              return;

       DiceManager.ClientAddDie(diceBlueprint,dieFaces);
}
```

Errors

If you get a strange looking error that refers to unity networking and can't figure out where it is coming from you may be trying to pass types that are not supported.

Network Messages

I use network messages when I need to send a message to a single client. You could use the RPC method above and check the client connection ID but you would still be broadcasting a bunch of data all the other clients do not need.

This process is a little more involved. I setup server and client master classes that will handle this network communication. These are again prefabs with your own network script attached to them. These prefabs do NOT need a NetworkIdentity component.

This type of communication requires registering handlers. The handlers are of type short and point to a method in the current script. I have a class with a number of constants for these.

Unity network messages are sent in the type MessageBase. You can add your own parameters that will be passed with the message.

```
public class NetworkMessageTypes : MonoBehaviour
{
	public const short SelectVector = 500;

	public class NetMsgSelectVector : MessageBase
	{
		public string boardGUID;
		public Vector2 targetVec;
	}
}
```

The following scripts are used to send a message from the client to the server. The client is telling the server that it wants to move a unit to a specific vector. First is the server script.

On the Master Server script you register handlers before starting the host. In the script below I am setting up a handler for when a user has selected a Vector2.

```
public class NetworkMasterServer : MonoBehaviour
{
	static public NetworkMasterServer instance;

	private NetworkManager manager = NetworkManager.singleton;

	void Awake() {

		if( instance == null )
			instance = this;
		else if(instance != this)
			Destroy(gameObject);
```

```
            DontDestroyOnLoad(gameObject);
        }

        //just a shorter way to access the Unity NetworkMarnager singleton
        private NetworkManager manager = NetworkManager.singleton;

        static public void InitializeServer(string ipIn, string portIn) {

            //interface field input for joining an ip and port
            instance.manager.networkAddress = ipIn;
            instance.manager.networkPort = int.Parse(portIn);

            //register event, NetworkServer is a Unity class
            NetworkServer.RegisterHandler(NetworkMessageTypes.SelectVector,
            instance.OnSelectVector);

            instance.manager.StartHost();
        }
}
```

This is the method that will handle NetworkMessageTypes.SelectVector. Unity network messages are sent in the type MessageBase. I am reading the base message out to my custom NetMsgSelectVector class which has the custom parameters.

```
void OnSelectVector(NetworkMessage netMsg){
        //messages are passed as the Unity type MessageBase
        NetworkMessageTypes.NetMsgSelectVector msg =
netMsg.ReadMessage<NetworkMessageTypes.NetMsgSelectVector>();

        //lets move the unit to a vector
        //unit to move: msg.boardGUID
        //move to vector: msg.targetVec
}
```

Why go to all this trouble? After a unit has moved the server needs to let the client that the move has occurred. The server can send a message to a specific client with this method.

```
 static public void SendSelect(int connectionID, string boardGUID, Vector2 vec) {

        NetworkMessageTypes.NetMsgSelectVector msg = new
        NetworkMessageTypes.NetMsgSelectVector();

        msg.targetVec = vec;
        msg.boardGUID = boardGUID;

        NetworkServer.SendToClient(connectionID,NetworkMessageTypes.SelectVector,msg);
}
```

Here is how the client receives the message.

```
public class NetworkMasterClient : MonoBehaviour
{
	static public NetworkMasterClient instance;

	private NetworkManager manager = NetworkManager.singleton;

	void Awake() {

		if( instance == null )
			instance = this;
		else if(instance != this)
			Destroy(gameObject);

		DontDestroyOnLoad(gameObject);
	}


	static public void InitializeClient(string ipIn, string portIn) {

		if( instance.manager.client != null )
			return;

		instance.manager.networkAddress = ipIn;
		instance.manager.networkPort = int.Parse(portIn);

		instance.manager.StartClient();

		//Register application messages
		RegisterClientHandlers() ;
	}



	static public void RegisterClientHandlers() {

		//after character move
instance.manager.client.RegisterHandler(NetworkMessageTypes.SelectVector,instance.OnSelectV
ector);
	}

	void OnSelectVector(NetworkMessage netmsg) {

		NetworkMessageTypes.NetMsgSelectVector msg =
netmsg.ReadMessage<NetworkMessageTypes.NetMsgSelectVector>();

		//after move on client
		//target vector: msg.targetVec
		//unit boardGUID: msg.boardGUID;
	}
}
```

Prefab Singleton Managers

I use a number of singleton managers. Too of these and your project can become difficult to adapt. There are arguments for and against such design but here is how to set them up. Create a script with MonoBehavior and attach it to the prefab you will be using in your scene. Here is the singleton part of the script.

```
public class NetworkMasterClient : MonoBehaviour
{
	static public NetworkMasterClient instance;

	void Awake() {

		if( instance == null )
			instance = this;
		else if(instance != this)
			Destroy(gameObject);

		//prevents the prefab from being destroyed when the scene changes.
		DontDestroyOnLoad(gameObject);
	}
}
```