Mitchell Fulton

Dr. Arden Ruttan

HONR-30096-031

25 April 2016

Study of Networking Techniques for Game Development:

The main goal of this Honors Individual Work assignment was to learn about the core principles of networking in regards to game programming. And then implement differing techniques of networking learned to continue development of a physics server. The server in question was created by; Zhuoran Wang, Chen Zhao, Tianyi Jiang, and Cheng Zhang, for their Game Development Practicum project in the fall of 2015. The goal of the continued work on the server was to see if changing the medium or techniques by which the networking was performed in the server would yield any comparative advantage or disadvantage compared to the original implementation submitted in 2015. All of this was done in the hopes of finding ways to improve performance on networked systems, while reducing latency and overhead.

# Basic Principles of Networking

To begin with we must first define networking in terms of game development. Basically networking is the process in which multiple distinct machines all running some version of the same software are able to sync the state of each machine across all machines. So actions taken on one machine are relayed across all machines connected to a given network. Thus allowing multiple users to experience their own distinct viewpoint in the same game as everyone else. Giving users the opportunity to effect and be affected by other users in the virtual environment. There are two main networking techniques that form the structure of all multiplayer games. Each with their own advantages and disadvantages, and each is best suited to specific types of games.

*Peer-to-Peer.* The oldest and simplest model for networking games is referred to as Peer-to-Peer or more colloquially P2P connectivity. The concept of P2P isn’t at all unique to video games and is widely used in the programming world. In essence P2P networks operate on the premise of interconnected nodes. Wherein each machine (peer) is capable of communication with any other machine (peer) on the network. Thus the name Peer-to-Peer. In regards to games, P2P works by having each machine run it’s own local version of the game. Whenever the user gives input that causes a change in the game environment this data is then relayed from the machine on which it originated to every machine on the network. For example if you had a set of connected machines and each machine was responsible for pushing a single block on a plane populated with blocks. Whenever one machine pushed its respective block due to input from the user, it would send a message to every machine on the network to apply x amount of force to said block. This model is very simple and fairly easy to implement. However it does have a few drawbacks that make it unadvisable to implement in many situations.

There are many issues inherent in this model but the problems boil down to four main points; security, synchronization, latency, and performance. Since each machine must be authoritative (meaning that the code on the machine has the authority to send commands to other machines and have them executed), it is a simple matter for even a rudimentary programmer to hack their local copy of the software and thus gain the ability to unfairly manipulate the result of online games. It is nearly impossible to stop users from hacking and disrupting multiplayer matches in the P2P model, and security in general is difficult to manage. This can put the average user of a piece of software in danger of other more nefarious users who may try and take advantage of the network to access the machines of clients connected to it. The other main problem for P2P networking is the latency inherent to this form of networking. Since each machine must wait for all of machines on the network to provide input in order to have a uniform simulation, P2P networked games often have latency that is equivalent to the speed of the slowest machine on the network. In addition the more machines connected to the network the more time each machine must devote to sending and receiving information. This makes P2P networking inadvisable for games with substantial numbers of players or if the connection speed of certain players is in question.

The third problem with P2P networking is synchronization. Since each machine runs it’s own version of the game while receiving messages about the objects in the game controlled by other machines, it is very easy for the machines on the network to become desynced. Take for example the box game discussed earlier. Say that two different machines receive the same command to apply force to a box. Even though the two machines are running nearly identical data on the same engine it is possible for the two objects end up in slightly different positions on each machine. As tiny differences between two objects on different machines only grow larger and larger as time progresses. Ultimately ruining the game when the whole network degrades into a confusing mess unique to each machine. Synchronization issues aren’t unique to P2P setups but P2P setups are typically the most vulnerable to them.

The final problem with P2P is the strain it puts on each individual command machine. Since each machine must run it’s own version of the software and receive and send commands. This means that each machine’s must have a certain degree of specifications so that it is capable of running the software.

*Host-Client.* The concept of a Host-Client network like P2P doesn’t originate from video games either, but due to its numerous advantages over P2P networks it has been widely adopted by the vast majority of video games since the early 90’s. The basic premise of a Host-Client network is that each user has stripped down version of the software on their local machine. The degree to which the software is stripped down varies but in terms of a pure Host-Client network the client machines can do little but send and commands and receive data. All of the processing for the simulation is done on the host machine. In this setup only the host machine has authority over objects, and it is solely responsible for all calculations. The client machines get input from the player, send it to the server, receive data from the server, and then display the data.

The Host-Client setup eliminates all of the problems with P2P, and comes with very few drawbacks. Since no client machine ever directly communicates with another client machine the security problems of P2P, are solved. Also it is significantly more difficult to hack the game because, not only is the majority of game code is not only absent on local machines, but also because the server is authoritative. Meaning only the calculations done on the server, according to the rules of the server will be accepted. In addition to this latency is much improved since the speed of the network for a given client is equivalent to the speed between that client and the host and not equivalent to the slowest client on the network. The Host-Client structure is by its very nature resistant to synchronization errors as well. Due to the fact that only one set of calculations are ever performed. In terms of performance however Host-Client networks are a bit of a mixed bag. While the performance demands for clients are greatly reduced, the performance demands for the machine running the server can be increased since it must handle all calculations and all network traffic. Another issue is that a dedicated server must be maintained in order for the users to be able to play the game, which can become quite costly. While Host-Client is preferable to P2P it is still far from perfect. The latency is significantly lower on average, however for most network’s (anything other than a LAN) the latency is still far too high to deliver a playable experience. Since messages still have to be relayed back and forth after the player provides input for a change to occur on a local machine.

*The Great Compromise.* There are a couple ways to decrease latency to make a game playable on slower network speeds for both Host-Client and P2P networks.. Firstly minimize the network traffic as much as possible. Sending less data, less frequently between the host and clients (or between clients in P2P) can greatly reduce the time a given user has to wait. Unfortunately culling erroneous network traffic only works in networks where bandwidth is in question. That is to say that is approach will not improve LAN or high speed networks. Also the amount of network traffic can only be reduced to a certain degree. In every game there is some data which absolutely must be sent at certain times. This approach does little more than alleviate the symptoms (high latency) of the networking models, since the clients will still be idling while waiting to receive data from the host (or another client for P2P). So this approach to improving network latency is limited in it’s effectiveness in a number of ways.

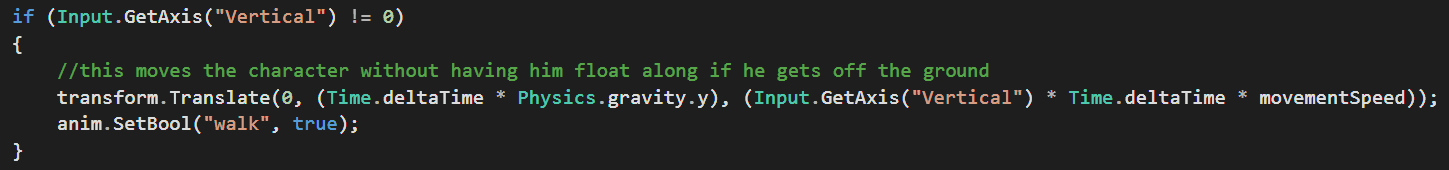
The second solution to improve networking functionality is to combine some aspects of both P2P networking with the Host-Client network structure. Each client will run it’s own version of the game (physics, etc), and everytime the client machine receives input from the player it will update the local simulation and also send a message to the server notifying it of what happened. The host takes messages from all clients and runs it’s own authoritative version of the simulation. So every client and the server will be running simulations in parallel. And at fixed time intervals or on certain events the host will transmit relevant data to all clients connected. The client machines will check to see if their local simulation matches the data that the host transmitted. If not the client simulation data is either discarded or modified to match the host data. This way the average user will see a result on screen noticeable faster than the other networking models since their local machine will immediately start running code after receiving input. In this hybrid network model synchronization is a practical non-issue. This being due to the fact that both the client and the host are running the same code on the same data so the differences between results in the simulation will be marginal at worst. Any differences will also be quickly corrected by the host sending data to the client to fix any micro desyncs that may have occurred before they can grow large enough to negatively impact gameplay. Thus a hybrid model can effectively solve the problems of latency in both networking models, while preserving most of the benefits of the Host-Client model.

# Original Server Design

The initial server project was created using a combination of Unity and SmartFox. With SmartFox handling the communication between the machines running the game and Unity handling the all other aspects. Such as rendering, audio, animations, and physics. The server was built around the Host-Client structure. The server features two different Unity projects, one for the client side and one with the server side. The host side machine is authoritative and takes input from the client machines and handles the physics collisions. To increase performance the host machine has all rendering and other non-physics calculations disabled. Also a SmartFox server must also be hosted for the the physics server to work. Ideally this is run on the same machine as the Unity host but it does not have to be. Here is the basic sequence of events that occur when a user wants to join a multiplayer game; The client machine activates and sends a message to the SmartFox server. The server receives the message then sends another message to the hosting Unity machine that a player has spawned. The host machine then executes code to register the user in the database and to spawn a player object from it’s prefab into the game world to represent the client machine that just connected. The majority of interactions follow this same basic structure. Something happens due to player input, message sent to server, server relays to Unity host, Unity host executes code appropriate for situation. To improve performance the previous team implemented a full simulation on client machines as well. So on player input the local simulation takes action while simultaneously sending the message to the SmartFox server.

The main discrepancy in this model is when the host (because it is authoritative) must send out it’s updates to client machines. This occurs in the function FixedUpdate() on the host machine. Which is called everytime a physics interaction occurs. In this case the host sends a packet of data (mostly positional info) to each client on the network. The client machine then compares the received info with it’s own simulation info and fixes any errors. Usually by interpolating positional differences if they are minor or snapping the objects to the correct location if the errors are large enough to have a profound and immediate impact on gameplay.

On the subject of gameplay, let’s summarize the various features provided by the previous team. A more detailed description can be found on the universities svn database (link in bibliography). The game has player robots that spawn on player connect and are capable of walking around, and interacting with the environment to a limited degree. They move decently and all have a working set of animations. Unfortunately there is a bug which causes the player object (aka robot kyle) to be immune to the effects of gravity while moving. So if he walks over a box without stopping, the player object will rise a little above the ground and remain there even after passing the obstacle. This is due to a simple error in the movement code that translates the player objects transform by increasing or decreasing the x value of the player’s transform while keeping the y and z values the same as they were at the start of the function. Thus height can only be changed due to collisions with other objects in the scene and not gravity. I fixed this bug in version 4.0 by simply adding a change to the transforms y value to reflect the force of gravity into the function that moves the player transform.



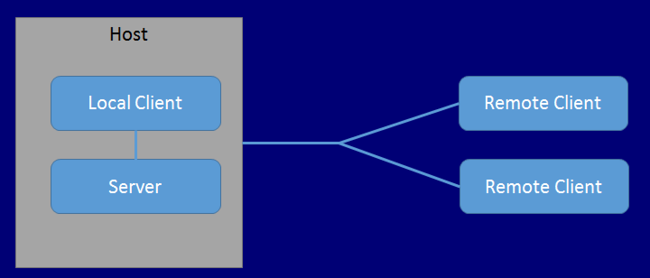
The other features of the game included a car which was drivable (with working animations for driving and physics for the wheels), and the ability to pick up boxes and bombs (also with animations). These items randomly spawn on the playing field and the bombs will explode dealing damage to nearby players.

# SmartFox

SmartFox is a “massive multiplayer SDK for building games” which is a fancy way of saying SmartFox provides a way to send and receive messages between applications on different machines, an intermediary whose main job is to facilitate communication. SmartFox follows the Host-Client structure for networking. Where the SmartFox host (the application running on a given machine) is responsible for receiving messages from all clients and then deciding what to do with it. SmartFox does come with a lot of inherent features that make it very practical for game development. In their own words SmartFox offers a “Rich client and server Framework with dozens of functionalities out-of-the-box: Session management, Room management, chatting, advanced lobby and Game Matching features, persistent Buddy Lists, moderation, server Variables and lots more.”. Perhaps the best part about SmartFox is the fact that it can easily be extended to meet any requirement a game programmer can think of by simply adding Java code into the server.

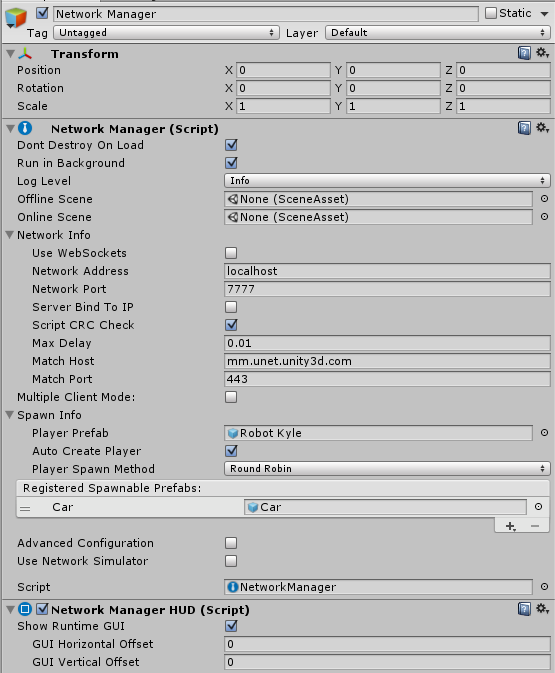
# Unity Networking

Much like SmartFox Unity Networking also follows the Host-Client Networking model, albeit with some modifications. Each Unity project that has networking enabled can function either as a client or as a host.

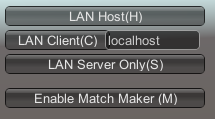


The model for Unity Networking provided by Unity is shown above.

Every Unity Networking project must have an object that acts as the “Network Manager”. This is a prebuilt tool that handles local play between instances of the software on a LAN and coordinates with Unity’s servers for online matchmaking.

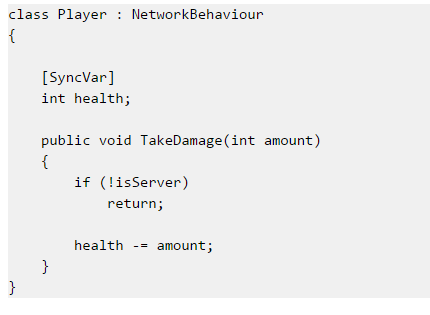


On startup the manager will prompt the player through a GUI whether or not they want to host or join a game and which type of multiplayer they would like (LAN or online).



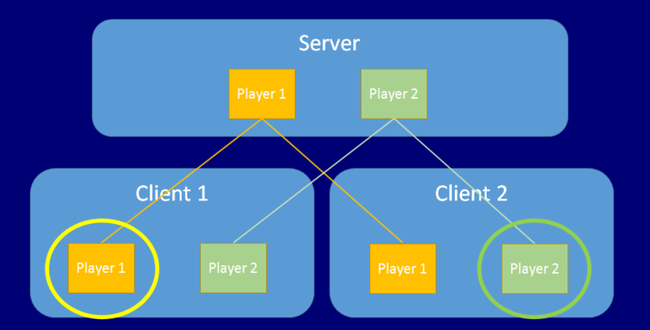
If the user expresses a wish to join a game using LAN then the network manager finds any unity instances that have the same project identification credentials as itself on the local network. Online play is typically routed through Unity’s matchmaking services. To access these services the author of the project must register their Unity project id credentials with the Unity online multiplayer services website. If the project has been properly registered, then the a local copy of the software can connect to Unity’s servers which will then relay network traffic to another machine that is running the same software but who has elected to host. On the subject of hosting, Unity does things a little differently than the typical host-client model. If the player elects to host a game, the player receives the same experience as other clients except that Unity will be handling the role of hosting in the background. Essentially choosing to be a host in Unity networking means that you become a client connected to your own machine. In addition to managing network traffic the network manager has a few other key responsibilities. Most notably being spawning player objects on connection from a client, and handling scene selection on startup. Programmers also have the option of overloading the network manager's functions to meet their needs for custom functionality.

For a multiplayer game to work, objects have to be synched across all connected machines. Unity has an incredibly easy way of accomplishing this. To change a gameobject so that it is regularly synced on all machines, one only has to add a network identity, and a network transform. Animations can handled through a network animator, which provides similar consistency for animations.



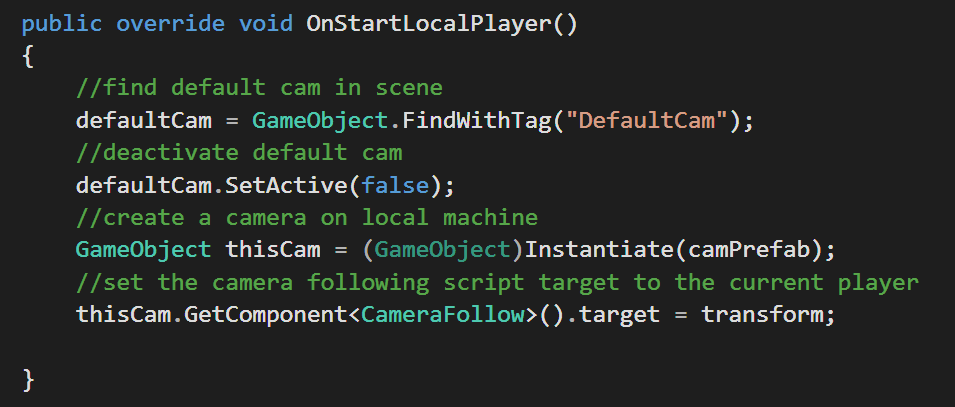
Network animators, transforms, and identities are all standard Unity components that can be added by a mere handful of clicks. A network identity allows Unity to differentiate between objects and associate properties with it. A network transform is one such property. This property gathers positional (or in some cases physics) data on a given gameobject and then relays this info through the network at set intervals which can be adjusted by a variable in the component editor. Network identities are also integral in determining which machines have authority to affect gameobjects. If there is important data other than animations or positions that should be consistent across the network one can use sync states. Here is an example provided by Unity of a possible use/implementation of sync states.

Unity also has a relatively distinctive method for handling authority. In a sense it subscribes to the Host-Client model where the simulation run on the host’s Unity engine are authoritative and override local simulations. But there is also authority in terms of which clients can move which objects. Meaning the calculations run on the server are still authoritative but only the machines with authority over a given object can tell the server to perform calculations. Since networked objects are physically present on in the Unity scene of all clients connected, some rather nasty and confusing errors can occur. For example say two players connect as clients to a game. The network manager spawns a player prefab for each of them. Each player prefab contains a simple script to move the player object around on input from the keyboard. However since both player objects have the same script, and exist in the scene, they both move when a single user provides input. This problem can be corrected by altering variables in the network identity components of objects specifying if local players can have authority and putting checks in scripts when necessary to test the origin of input. Unity networking is easy to implement if a little hard to wrap one’s head around.



Here is the visual representation provided by Unity.

One of the benefits of Unity Networking is that programmers can decide which objects to network and which not to network. This can provide numerous advantages but also can become quite the headache. For example in the previous scenario with two connected players. How does one handle camera work in a situation such as this? Where multiple players are present and each requires their own camera. The way I handled it was to have the player objects be networked and then attach a script to each one that spawns a non-networked camera to follow the player object associated with the user. This way the scene has no problems with multiple cameras being present, as each client's camera will only exist on their local machine. This was accomplished by overriding the function OnStartLocalPlayer (function that is called when a player connects) and putting the code for creation inside of it. However if you take this approach make sure to clean up after yourself adequately. If for example you disconnect and reconnect without restarting the game, you will have two cameras on one machine. Make sure to carefully manage objects that only exist on local machines.



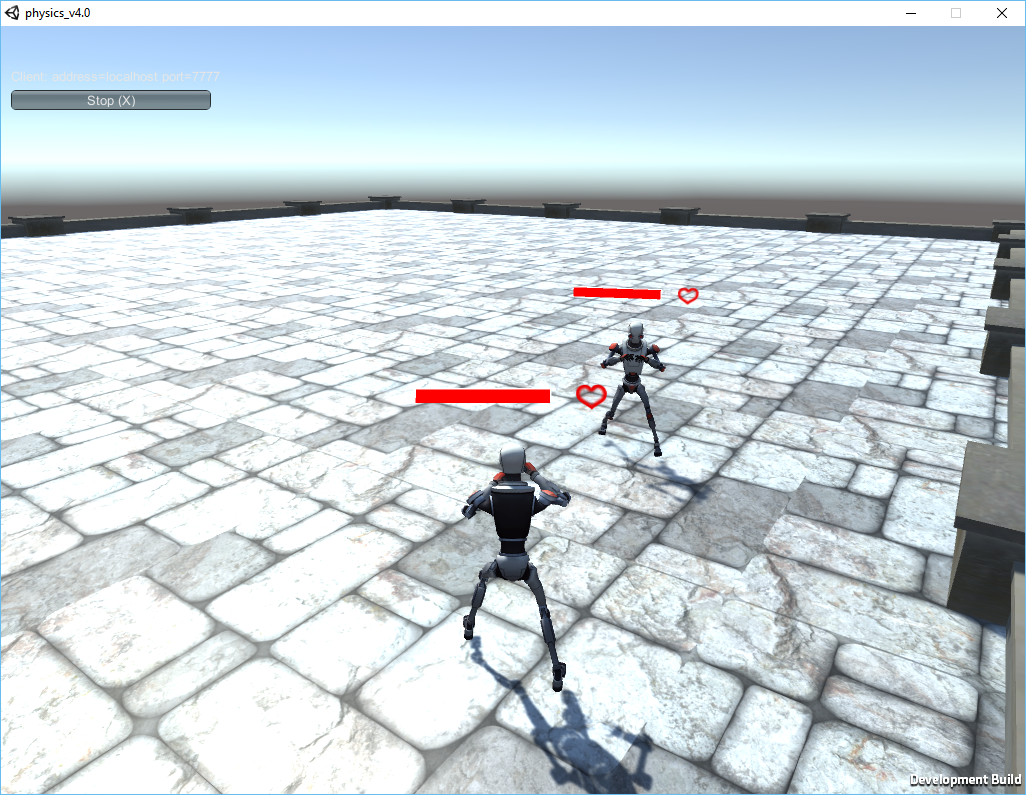
Before continuing we should briefly discuss commands and rpc calls. The specifics and how these were used in the project update will be discussed later in this paper. The unity networking tutorial describes these perfectly as “ClientRpc calls can be sent from any spawned object on the Server with a NetworkIdentity. Even though this function is called on the Server, it will be executed on the Clients. ClientRpc's are the opposite of Commands. Commands are called on the Client, but executed on the Server. ClientRpc's are called on the Server, but executed on the Client”. Commands and Rpc’s are Unity’s built in way of having servers and clients communicate. Unity Networking also provides ways of sending direct messages in between machines. Allowing a similar implementation of networking to the original project. However in most situations using commands and rpc’s are more convenient and more powerful. Where messages are sent on certain events and the server parses them and executes code. Commands and Rpc’s have limitations and often require some round about thinking, but they provide a valid alternative to raw message delivery.

# 

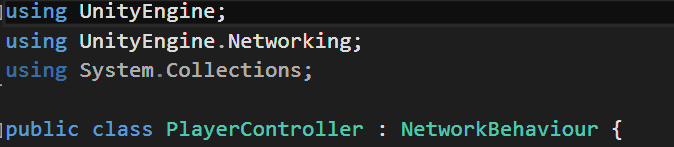
The above chart provided by the Unity documentation provides an excellent visual aid to understanding Unity Networking.

# Current Implementation

So far this report has focused on very high level concepts on the nature of networking in general, and how Unity Networking and SmartFox operate on principle. In this section I will strive to explain in detail the process I went through when implementing Unity Networking into a project. I will keep it general enough so that this outline will be useful if anyone in the future tries to use it as a guide, but also specific enough to showcase my work on this project. Detailing the steps I took and the major problems or misunderstanding that held me up along the way. The previous team’s last version of the project was a client side version 3.1 and serverside 2.0 (these can be found on the kent svn site and in the provided code with this report). Due to the previously discussed nature of Unity networking I had to combine these two versions into a single project when updating it to use Unity networking as opposed to SmartFox. Thus physics version 4.0 was born.

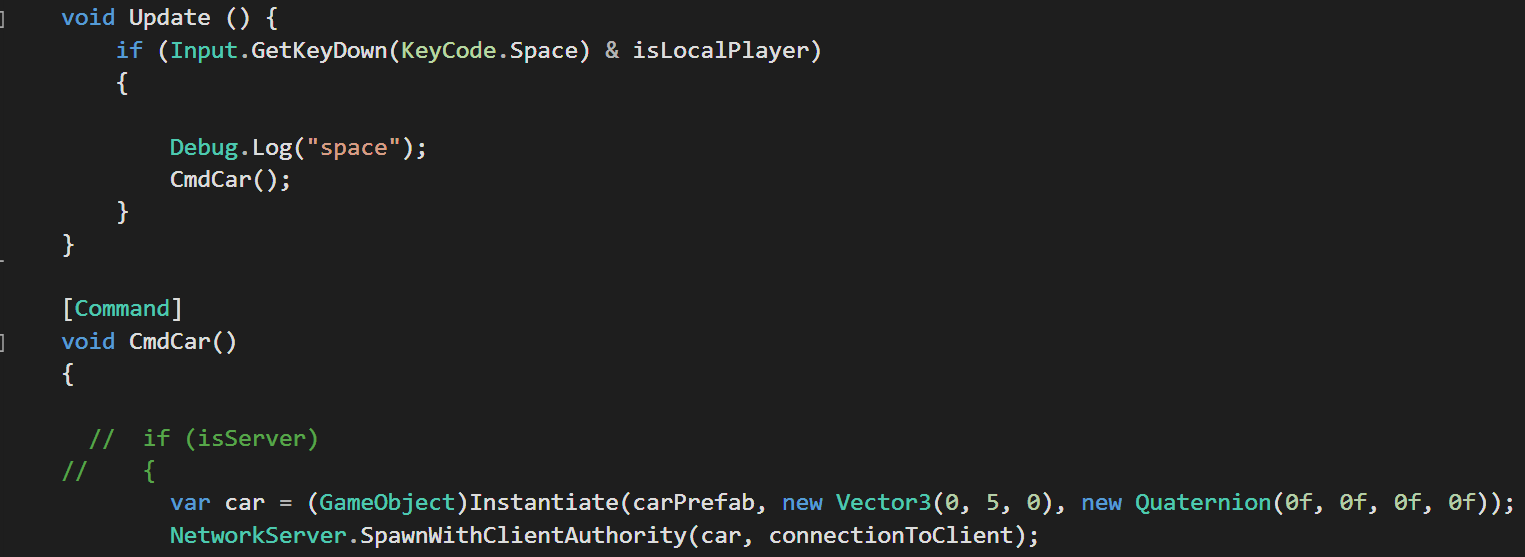


We have already covered a lot of the basic aspects earlier in this paper, but first things first there are a couple things one must know to be able to correctly script with Unity Networking. First of all much like when using Unity’s built in GUI systems any scripts written for networking purposes must have using UnityEngine.Networking;, in their include sections. Also the class definition must be changed from MonoBehaviour to NetworkBehaviour in most cases.



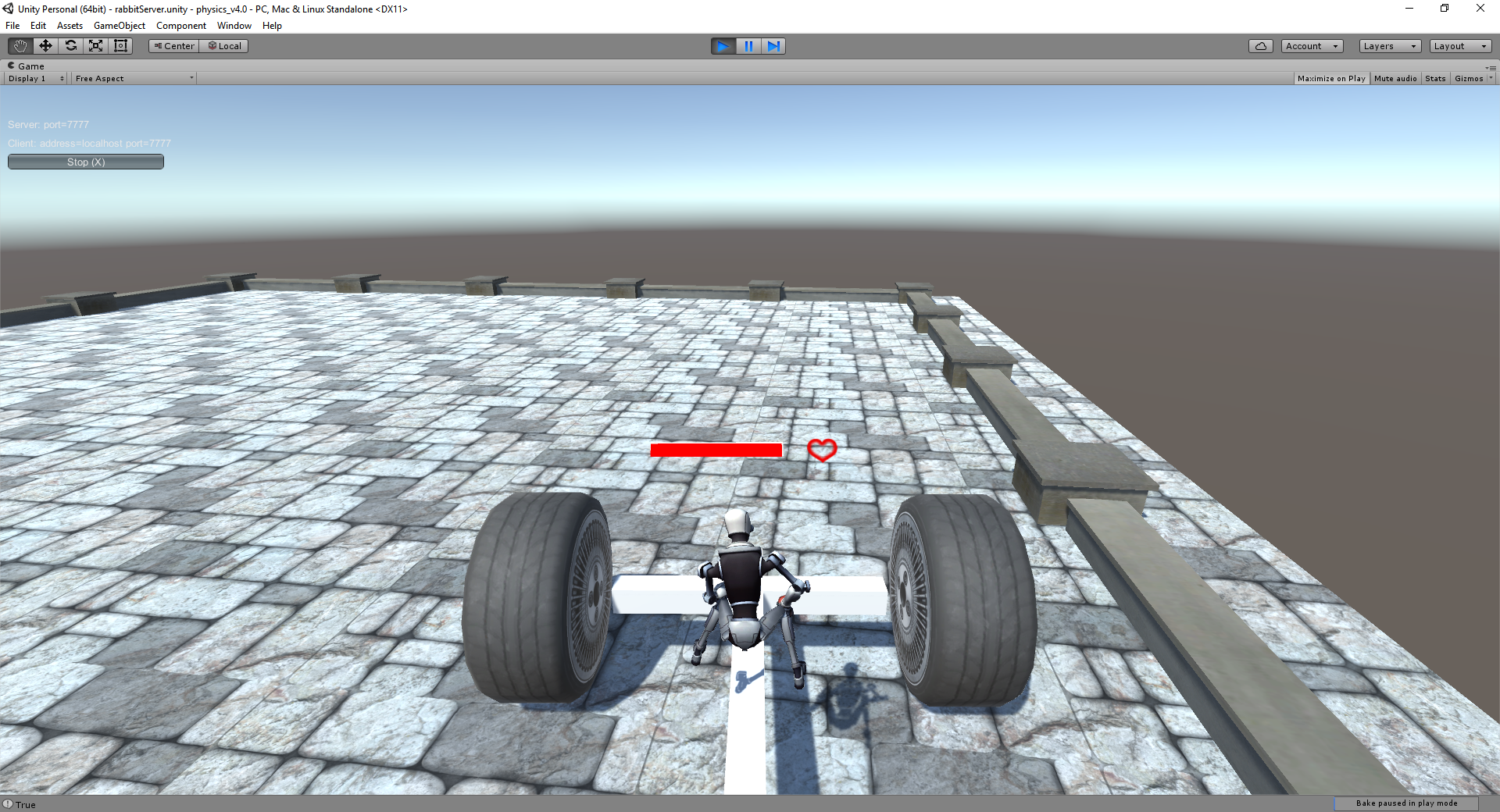
These two changes must be made so that the integrated development environment knows what’s going on when you begin writing code that pertains to network aspects of Unity.

I would say that the biggest problem I faced when updating the project was getting the car to work correctly. In addition the example of the car is an excellent means to walk through many aspects of Unity networking and some common solutions to common problems. Let’s begin with spawning the car itself. Any object that you wish to spawn in game must first be registered in the network manager. This is relatively simple as it just involves taking the prefab and adding it to the list of spawnable items. Once it has been registered the machine that wishes to spawn the object must send a command to the host. Commands are essentially functions that when called send the code inside of themselves to be executed on the server.



Unity networking is still fairly new and doesn’t provide great functionality for complex physics objects. Basically here are some rules to follow with complicated game objects that involve physics and or have a lot of child objects. Attach a network identity and transform to the parent object as usual. Then add network transform children to the parent object as well and then set the target of said component to each of the children. Do not attach individual network ids and transforms to each of the children. Doing so will in this case at least cause the car to violently explode into pieces and go soaring into space. I’m not particularly sure why this happens, if I was to guess I’d would say that the micro movements in the transforms of the children inside the parent due to the network manager trying to update their positions was the cause of the issue. Also when using Network transform one can choose to sync the rigidbody or the object's transform. The correct one to use depends on how the object in question is planned to be manipulated. For example if you have an object that needs to move upon player input. If the object is moved by altering its transform it is better to use sync transform. If the object is being moved by applying forces to it, then it is better to sync the rigidbody. This is at least true in some cases. However I found that during the testing I performed during this project that in all cases syncing the rigidbody provided more consistent and smoother performance. My advice would be to try it both ways (assuming the object has a rigidbody) and see what works best.

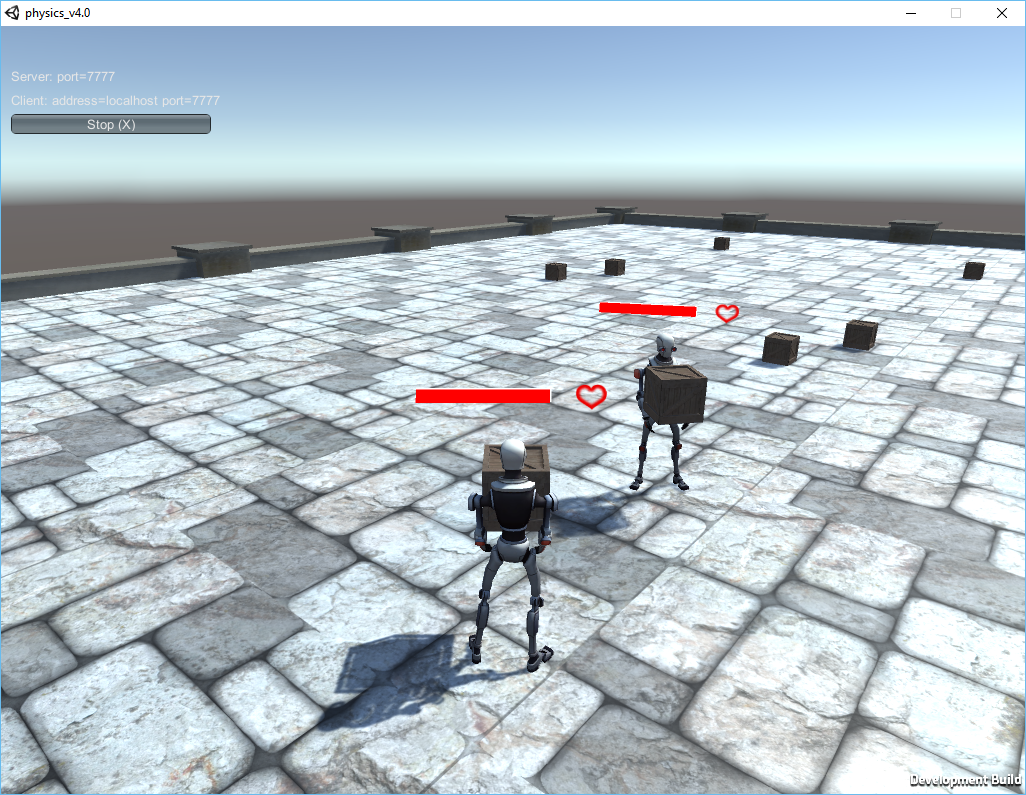
Another problem experienced by the car is that it is a non player object that must respond to player movement. See when a client connects and a player is spawned from a prefab, the most recently connected client is given authority over that game object. Not authority in terms of Host-client but authority in terms of it being the only one to make changes to the gameobject. This works well for the player objects but can be tricky when one wishes to implement more advanced objects such as vehicles that can entered and exited. The only way to have a user be able to drive the car is to have whatever machine that wishes to control the car have authority over the car. The best way I found to do this was simply to have whatever client that decided to spawn the car get authority over it using the spawn with client authority function in a command sent to the server. This setup means no one can steal your car, and it means that car movements are accurately relayed across the network. Unfortunately it also means that no one else can use the car due to limitations of the design. This limitation was overcome and implemented for how the player interacts with the wooden crates in the game. However I figured for the sake of comparison and to keep people from stealing cars I would retain the aforementioned technique for managing the cars.



When a player wishes to pick up a wooden box the local machine sends a command to be executed on the server that assigns authority for a pre existing object in the scene. This is different from the car in the sense that the car was spawned with authority. This assignment is accomplished via acquiring the network instance id (a uniquely generated key that each object that has a network identity acquires on game start) of the box the player is colliding with, and then passing it as a parameter of the command. Along with the network identity of the player trying to pick up the object.



This command is necessary because while authority can be assigned to any client for any non player object, it can only be done on the server. So the network instance id allows the server to find the correct box and the network identity of the player allows the server to find the correct client to give authority. Once this command has been sent the implementation becomes similar to the car. Where local code is run to determine the box position according to player movement while commands are being sent to the server that run similar code to ensure responsiveness and consistency across all machines.



On the topic of the wooden crates, let’s discuss how to spawn them. Spawning objects can be achieved by attaching a script that does the spawning to an empty game object. Then setting the network id of said game object to server only. Meaning that this object will only exist/execute on the server. This is very useful for tasks such as spawning where it is desirable to have one central uniform system. Because if a copy of the spawner was operating on each client then the maximum number of objects that can be spawned in the scene, and the rate at which they spawn would be dependent upon the number of clients connected currently. Having the spawner only run on the server eliminates that problem. Be very careful of how you assign the properties of a network identity. Some objects like spawners should only run on the server, but other objects like a crate must have a local player authority option, otherwise they cannot respond to client input under any circumstances.



# Comparison of Techniques

SmartFox is more flexible and easier to expand in general. Unfortunately it also has a significantly higher barrier to entry when compared to Unity Networking, which can get a project running in minutes. SmartFox also requires a separate application to be running at all times for use, along with specialized plugins to Unity for everything to run. In addition, to use SmartFox one must be able to code in Java. Not a huge hurdle but one nevertheless for some who might be used to only using C# or another programming language. Unity Networking has a more limited basic functionality in comparison, but it is also far simpler and more user friendly. Mainly due to the fact that everything is done in Unity itself and the majority of networking can be accomplished by adding or altering objects in the Unity Scene editor as opposed to hard coding things. Unity takes care of most of the nitty gritty aspects of networking on it’s own, letting the user focus their attention on other matters. Unity’s ability to serve as either a host or as a client does simplify distribution, but it also reduces some freedom of development over the software. The online service Unity matchmaking is great for getting a project off the ground in a matter of minutes, but it is currently the only available solution for online play using Unity’s networking features. Also it is technically possible to customize Unity to suit specific needs, but this can be difficult especially considering the lack of decent documentation on Unity networking at present.

In all of my personal testing I didn’t experience any issues with online matchmaking using Unity servers. Running multiple instances on the same machine (over LAN) or on multiple machines off Kent’s wireless didn’t present any issues. However when I provided test builds to some of my friends who live in Columbus we were unable to get a stable connection. This may or may not be an isolated case and could be due to their networks and not Unity. While I didn’t have time to test extensively nor to debug the issues experienced, the likely cause is that Unity Networking implementation requires a stable and faster internet connection than the previous SmartFox implementation.

One can technically choose a host that isn’t Unity matchmaking by changing the host address in the network manager, but this would most likely involve integrating with another networking platform such as SmartFox and essentially defeat the point of Unity Networking. Since most of it’s built in features are mainly designed to take advantage of not having to use third party software. In addition managing non player objects is overly difficult and it can be unnecessarily painful to debug issues due to the multi scene nature of Unity Networking. Unity’s built in state synchronization saves a lot of time as well and can be quite useful, however it is fairly rudimentary. Jittering, frequent position snapping, can be quite common especially when dealing with non player objects. Again Unity’s lack of well defined documentation isn’t particularly helpful at alleviating these problems either. Overall Unity Networking is good for those who want simplicity and ease of use over functionality. A good introduction for students and new programmers to get their feet wet in the world of game networking but not terribly useful for anything else. It has a lot of potential but isn’t terribly useful overall. When creating a professional project or one that requires more functionality SmartFox is probably the better solution as of the time of this writing. Of the two versions that now exist of the physics server project I would say that the original is currently the better implementation. (Note: Unity networking and the engine itself is constantly being updated, so this assessment may or may not be accurate for future reference)

# Further Ideas:

The majority of the game remake is done, however I was unable to completely finish implementing all aspects of the game. Let alone polish and stress test it. Here are some notes and suggestions if the reader wishes to finish or update this project further. Debugging why the game crashes on slower networks would be a good update. It is likely that error handling code needs to be reduced or the rate at which objects sync needs to be adjusted. (high sync rates can cause high network traffic and subsequently crash the game). The main unfinished issue is that of the bombs and player health. The original game had bombs that could be picked up, thrown and subsequently detonated to inflict damage on other players. Unfortunately I did not have time to implement this feature, but I do have some ideas for how it should be accomplished. Using similar code to the wooden crates would be an ideal way to add bomb functionality into the game. Player health (the ui is in place and working now) can be implemented using the example discusses earlier with state synchronization of variables. As for death and respawning Unity tutorials have a whole section on this but I’ll boil it down to a couple sentences. In essence use Client rpc calls to destroy or reset the player object on all client machines. Example from Unity shown below.



# Debugging and Design Tips:

Over the course of my time with Unity Networking I’ve learned a few basic principles about getting it correctly implemented. What follows is a list of common errors and suggestions that I have formed based on my experience.

* Always test on a server, on a client, and then test on a server and client simultaneously. Code that works on a server may not work on a client or vice versa, due to how authorities handled. In addition code that might work when only one person is connected may not work when multiple people are. I.E. having a single player's inputs controlling all objects in the scene.
* For networked objects always experiment with syncing the transform or syncing the rigidbody. One will invariable work better for the purpose intended.
* When in doubt sync the rigidbody.
* Clean up any spawned items on local machines, so that the project will still work if you want to reconnect without restarting.
* Code that is sent to be run on the server via commands, should also be executed in parallel on the client. This increases responsiveness and prevents errors from occurring in case the local object has been granted authority.
* If code isn’t working or objects are getting out of sync on multiple machines then authority probably hasn’t been properly assigned.
* Put network identities on pretty much everything.
* Use the network instance id to track objects across machines.
* Build checks into code to make sure that code runs on appropriate machines. I.e. islocal player.
* Check the network identity component of the object if you are having problems. You may have set it to be server only or client only.
* When working with attached objects especially those with complex physics use the child network transform

# CONCLUSION

All in all I’ve learned a great many things during the course of this honors individual work assignment. Namely the basic principles of networking, network models, the basics of how to implement said models, and the annoyance of debugging networked systems. If you are looking for a very specific breakdown or step by step tutorial on how to integrate SmartFox or Unity Networking I highly recommend the Konsnos and Unity Multiplayer tutorials respectively. They are both excellent articles that feature step by step basics to getting a game up and running. Or if you are just interested in game networking or development in general I would recommend the articles listed below in the citations section. Most notably the Gaffer on games website. They have excellent articles on most aspects of game development.

# Works Cited

The following aren’t works cited per say. This is a reference sheet for all of the articles and tutorials I could find (that were actually useful), regarding online networking in general and specific networking platforms discussed in this article.

<http://gamedev.stackexchange.com/questions/67738/limitations-of-p2p-multiplayer-games-vs-client-server>

<http://forum.devmaster.net/t/networking-p2p-vs-server-based-online-multiplayer-games/16869>

<http://gafferongames.com/networking-for-game-programmers/what-every-programmer-needs-to-know-about-game-networking/>

<https://svn.cs.kent.edu/gpg/trac/wiki/PracticumF15/physics>

<https://en.wikipedia.org/wiki/Peer-to-peer>

<https://konsnos.wordpress.com/2015/05/10/create-a-multiplayer-unity3d-game-with-smartfoxserver/>

<http://docs.unity3d.com/Manual/UNetOverview.html>

<https://unity3d.com/services/multiplayer>

<https://unity3d.com/learn/tutorials/topics/multiplayer-networking>

<https://www.youtube.com/playlist?list=PLwyZdDTyvucw5JhBMJxFwsYc1EbQYxr0G&feature=iv&annotation_id=annotation_3530634901&src_vid=NLnzlwCRjgc>