Sheffield Hallam University

Faculty of Science, Technology and Arts

Department of Computing

Project (Technical Computing)

[55-604708]

2019/20

Author:	**Thomas Dunn**
Student ID:	**26016614**
Year Submitted:	**2020**
Supervisor:	**Thomas Sampson**
Second Marker:	**Aaron Macdougall**
Degree Course:	**BSC Computer Science for Games**
Title of Project:	**Networking A Rigidbody Simulation**

Confidentiality Required?

NO ☒

YES ☐

Networking A Rigidbody Simulation

By Thomas Dunn

Please mark this work for content and understanding and not for structure or surface errors of standard written English (spelling, punctuation and grammar) except where these may impede meaning.

Contact Disabled Student Support on ext 3964 for further advice if required

Contents

2 Introduction

2.1 Abstract

This report will discuss the ways in which video games handle syncing a physics simulation over a network connection. This will interest any game developer who is looking into networking a video game even if it doesn't have a physics simulation these techniques can be applied to objects in the game world which don't get affected by the physics simulation but mainly focuses on a physics simulation as a physics simulation have unpredictable results which makes it harder to sync it over a network connection.

2.2 Introduction

Most modern video games have the use of a physics engine to make the game act more realistic in the regards to multiple objects colliding, this allows objects to act in a unpredictable way, like a ball falling off a roof, there are many places that ball could land, while this is amazing it does have issues when you are trying to make a multiplayer physics based game, because with the same state of the world on two machines the physics engine could potentially produce two different results down to the bit level, resulting in the simulation being different on both computers. This project will go over the different ways to sync rigid bodies on multiple machines.

2.3 Project Outline

The project outline is to network a rigidbody simulation, the ways you might go about doing so and potential problems which might come up when trying to sync a rigidbody simulation.

2.4 Aims and Objectives

The Project aims to see whether it is possible to network a rigidbody simulation and how you might go about doing that. For this project to be considered as a success the following objectives are defined

- Client/Server game so multiple players can connect (2-4 Players), with the objective of players interacting with each other and interacting with other entities in the regards to physics
- Implement several different techniques used to sync a physics simulation over two or more Computers
- Evaluate and compare the different techniques towards networking a rigidbody simulation
- Debate whether using TCP/UDP would be better for syncing a rigid body simulation using any of the techniques implemented
- Discuss which technique might be best in certain scenarios/network conditions i.e Type of game or game platform

3 Research / Literature Review

These days, games use physics engines to mimic real physics. Physics engines do a lot behind the scene to mimic real physics, like calculating collisions and how to resolve those collisions, as you can imagine these calculations are quite computationally expensive and have to run typically at 60 Frames Per Second and because of that, physics engines do a bunch of little things to keep them optimized, however these optimisations makes the physics engine act differently when it runs on different platforms.

There are a wide variety of online multiplayer games, all with their own requirements and differ greatly from each other. This also means that each game has a wide range of technical requirements that must be implemented, for example a real time strategy game, has completely different technical network requirements compare to a turn based game, for example a real time strategy could have a lot of units for the player's to control, sending all the position, rotation, velocity and angular velocity for every unit would cost a lot in regards to bandwidth, so a game like that might use a network technique to minimize the cost of bandwidth, which we will go in a bit more detail below. In this project I will be looking at the techniques used to sync a rigid body simulation.

When it comes down to making a networked video game, syncing physics over two computers isn't as easy as you might think. Physics engines typically use floating point to represent its data, as floating point is great at handling very small/high precision numbers, so this is great for physics. Another issue to consider is that computers use a system called floating point to represent and manipulate decimal numbers, which is an approximation to decimal numbers, as you can't take a number like 1/3 which is 0.333 recurring and put that into a computer, as that is in infinite number and a computer would run out of memory very quickly, hence floating-point arithmetic, some compilers will handle floating point numbers different to trade off precision over performance as having a huge number like 1/3 typically you tend to only care about the first 4-6 values after the decimal point making the rest of the values redundant. Because floating point numbers give different results depending on what compiler/hardware/architecture you are using (giving very slightly different result down to the bit level) when two computers use a physics engine even with the same input and the same state, you're not guarantee to get the same results on both computers.

A GDC talk called Game Networking for Physics Programmers (Glenn Fiedler, 2015), mentions that there are 3 techniques typically used to sync a physics simulation which are; Snapshot Interpolation, State Synchronization and Deterministic Lockstep each having their pros and cons. I will be experimenting with all three techniques in order to see the advantages and disadvantages of using them to see which technique best fits certain genres of games. These techniques seem to be the solid approaches to networking a rigidbody simulation in basically any game engine/framework.

The talk to Pat Wyatt at HandmadeCon (2015), mentions of the technique called "Deterministic lockstep", which they used in games like Warcraft. With Deterministic lockstep it's required for you to have a deterministic physics engine meaning with the same state and same input you'll get the same result down to the bit level, you could hash the results and compare them and they should be the same. Pat Wyatts (2015) mentions that the reason why they choose the lockstep approach was because it's cheap regarding bandwidth but has disadvantages of high latency. So, this technique would be good for a real time strategy where there is loads of units the players can control.

Age of Empires is an RTS game which gives the player control over several units to attack the enemy. As you can imagine controlling so many units would cost a bunch of bandwidths however at that time, they had very little resources to handle the number of units on the screen regarding

bandwidth. In the article Network Programming in Age of Empires and Beyond (2001) mentions that they use the lock-step approach to network Age of Empires “Since the simulations must always have the exact same input”.

One of the issues with most common game engines like Unity and Unreal is that by default they use NVIDIA PhysX which isn’t deterministic. However, looking into the Unity updates log it states Physics Changes in Unity 2018.3 Beta (Yakovlev A, 2018) that there’s a new thing added to the engine called Enhanced determinism Yakovlev (2018) the article writer states that “PhysX guarantees the same simulation result when all the inputs are exactly the same” this will be useful in my project as I’m using Unity 2019 which should include this update and that should hopefully mean I won’t have to swap out the physics engine for a deterministic physics engine. However, Yakovlev (2018) mentions “It’s not free in terms of performance, but it shouldn’t slow your project down significantly” which is something to be aware of as this indicates it does have an impact on performance, however with this project there will likely be no noticeable difference when compared to a larger, more complex game.

In an article about Deterministic Lockstep (Glenn Fiedler, 2019) when using deterministic Lockstep you only ever send the inputs as the physics is deterministic, so you need to ensure that the inputs arrive and, in the order that they were sent in, as Glenn (2019) puts it “Since we can’t simulate forward unless we have the input for the next frame”. We could use TCP to ensure that all inputs are received, however as Glenn (2019) states that “each time a packet is lost, TCP has to wait RTT*2 while it is resent” which results in hitches and will get worse the more packets that are lost.

Glenn (2019) mentions a solution for this “we redundantly include all inputs in each UDP packet until we know for sure the other side has received them”. This way we can get all the inputs from the client which the client knows hasn’t been simulated by the server without sending them via TCP. Reason why we need to have all the inputs is we require to have the next input to carry on with the simulation and if a packet is lost then we lose an input, this way it ensures we have all inputs and this works well as inputs are typically small, so you can get away with it.

Now that we have all the inputs we need to simulate them, however Glenn (2019) points out “When you send inputs over the internet, you cannot just take the input and run that frame straight away, there’s a thing called Jitter. You won’t get nicely spaced inputs, you’ll get clumps, you’ll get 3 inputs, no inputs, no inputs, one input and so on, you need buffering”, this is very important to my project as without buffering I won’t get a smooth result. If you didn’t do this some frames, you’ll have no inputs which means you cannot simulate that frame, until it arrives.

The article Networked Physics in Virtual Reality done by Oculus (Glenn Fiedler, 2018) explains another technique, which is State Synchronization. In this technique you run the simulation on both sides and just patch it up when it goes wrong, Glenn (2018) mentions “I fight non-determinism by grabbing state from the left side (authority) and applying it to the right side (non-authority) 10 times per-second”. As the simulation will be running simultaneously across multiple machines, it is not possible to perform any interpolation as that would affect the physics simulation from being a simulation as Glenn (2018) mentions “simply snap the position, rotation, linear and angular velocity of each cube to the state captured from the left side” passing the linear and angular velocity we can use that in the simulation to simulate/extrapolate where the object needs to be.

However that’s may not be enough to achieve a smooth result, in the GDC talk called Game Networking for Physics Programmers (Glenn Fiedler, 2015), he mentions about State Synchronization and that snapping the results hard isn’t always enough to achieve a smooth result

and that you should still snap the physics result HARD but also calculate the error margin and interpolate the graphics of the physics using the error. This is all needed to ensure a smooth visual result, as when making video games, game developers tend to try and simulate what you would expect in a real life scenario, so if you're game has hitches then this is a eye swore behaviour and gives unpleasant feel to any title.

An article by Glenn Fiedler called Snapshot Interpolation (Glenn Fiedler, 2014) states one more technique for syncing a physics simulation which is Snapshot Interpolation. Instead of only sending inputs like with Deterministic Lockstep, we send all the visual simulation elements (Snapshots) from the server to the clients, then on the client we use the visual data to interpolate between frames much like how streaming sites work Glenn (2014) "all without running the simulation itself".

However much like deterministic lockstep, when the client receives the snapshots it will not come through nicely spaced like it was sent on the Server, you'll get clumps, so we need some sort of buffering. Glenn (2014) mentions using a different type of buffer "This interpolation buffer holds on to snapshots for a period of time such that you have not only the snapshot you want to render but also, statistically speaking, you are very likely to have the next snapshot as well". So, in short you hold the snapshots coming in for a certain amount of millisecond so that there is some kind of buffer, and then start dequeuing the buffer at the same rate as the server send rate so that there should always be something in the buffer.

In the interview that I conducted with the Tiny Tanks developer Lukas Krebs. Tiny Tanks is a physics-based party game, released in early 2019. When talking to Lukas regarding how they managed to network Tiny Tanks without any noticeable hitches. Lukas (2019) mentions with in Tiny Tanks they do some interpolation as snapping the physics state hard causes jitter results, Lukas (2019) states that "soon as you the controller collide with an object I simulate local physics for you and the object your colliding with, and as soon as the collision ends I turn snapshots back on and interpolate to where the objects should be", so as you can see they utilize a bit of both techniques. This interview was extremely valuable as not only did they manage to simulation a physics simulation over the network they also managed to release a successful physics-based party game on Steam.

The GDC talk Replicating Chaos: Vehicle Replication in Watch Dogs 2 (2019), they mention they use a peer to peer architecture instead of a client/server architecture, so they don't have one computer in charge of the physics simulation. They mention they utilize snapshots/state synchronization hybrid, where they simulate the physics locally and receive snapshots from the other peers to alert other peers where their vehicle should be at this given frame. They do a lot of switching between extrapolation and interpolation depending on how far the time offset between snapshots are.

4 The Selection of Tools

The requirements of this project come down to two things, being able to render some sort of geometry while apply some sort of physics to that geometry and to be able to communicate data between two or more computers. As this project mainly focuses on networking a physics simulation over multiple computers, I don't think it's necessary for me to build a game engine but instead use something I can quickly put something together so that I can focus directly on the implementation of the synchronization of the physics simulation.

Game Engine	Built in Networking Solutions	Third Party Networking Solutions	3D Physics Engine	Good for rapid development
Unity				
Unreal Engine				
Custom Game Engine				
Game Maker				
Lumberyard				

Figure 1- Selection of Game Engines

So, after careful thought I've decided to use Unity 2019.1.8f1, because at the time of writing this, I am more familiar with Unity Game Engine then any of the above. One thing I've taken into account is Unity doesn't at the time of writing this have a fully stable networking solution, as they recently removed their networking solution UNET to replace it with a new updated networking solution to support all their new tools such as ECS (Entity Component System) and the Jobs System which at its current state isn't fully documented. As Unity didn't have this in place, I wanted to see whether with third party networking solutions you could still simulate a rigidbody simulation. I could of used Unreal Engine but I choose not to simply because my knowledge of Unreal Engine is lacking and for this project I wanted to focus on networking a rigidbody simulation and not whether I can use a game engine effectively, so Unity was the obvious choice, on the plus side Unity is great for rapid development.

Networking solutions was the next question in my mind with Unity's UNET now deprecated I had to search for a third party one. One of the requirements which I set is that the networking solution had to be a high level networking solutions instead of a low level transport networking solution, which high level basically is develop with Unity so that everything just works, so game objects in Unity will be synced so that you can send data from one game object to another, over the network, this will allow me to concentrate on the implementation of networking a rigidbody simulation instead of having to build a system to allow communication between game objects. Another requirement was the networking solution had to be free.

Figure 2 - Forge/Photon/Mirror

As for networking solutions I had a bunch to choose from, Photon, Mirror, Forge Networking and many more. Forge Networking is the one I'm most familiar with, it has a big community with active support, but one thing which is really good about Forge Networking and that it is Open Source, so if something wasn't working as we might expect it, we can patch that up, if needed. Forge Networking also handles all the syncing of Game Objects as well so we could focus on the main networking techniques for this project instead of having to worry about doing loads of basic UDP socket connections and syncing Game Objects so that they match on all connected computers.

As for other networking to choose from, Mirror looked quite ideal, it's based upon the deprecated Unity Networking called UNET, it has all the benefits of UNET with a lot of bug fixes and improvements, the only downside I could find is it's very high level, in regards it hides all the networking behind attributes so it can be quite hard to figure out what is going on behind the scene.

Networking Solution (Third Party)	Is High Level networking solution	Supports UDP	Supports TCP	Relies on external service	Is Open source	Is Free
Photon Pun	✓	✓	✓	✓	✗	✗
Photon Bolt	✓	✓	✓	✗	✗	✗
Forge Networking	✓	✓	✓	✗	✓	✓
Mirror	✓	✓	✓	✗	✓	✓
LiteNetLib	✗	✓	✗	✗	✓	✓

Figure 3 - Selection of Third Party Networking Solutions

4.1 Revenue Considerations

Another benefit of using Forge Networking instead of using Networking Solutions like Photon is cost, I wanted to use a free network solution in terms of everything is free because when making a video game you don't really want to pay a CCU cost on top of other fees i.e Steams 30% cut or Epics Store 12% cut as shown in Figure 4 - Store Revenue Split, especially for developers like myself who want to release a game on Steam and to utilize their P2P Networking, which is fully free for all Steam Developers, so Photon really didn't fit our needs, as even though Photon is free it does cost once you start getting players onto your server CCU (Concurrent Users). To follow up on this I did email the developers of Photon to confirm if I wanted to utilize Steam P2P networking in Photon would I still have to pay for CCU and they responded with "you would need to buy commercial licenses" which completely took out the word free in my case.

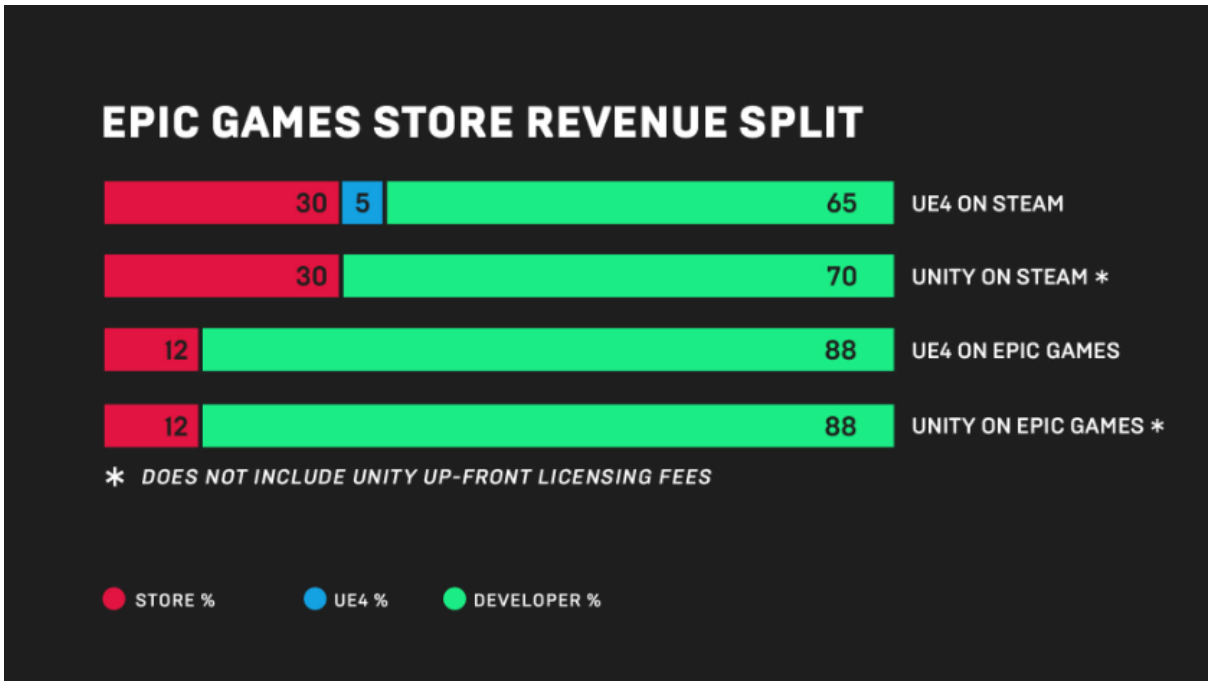


Figure 4 - Store Revenue Split

4.2 Clumsy

Clumsy is a networking simulation tool used to simulate certain network conditions, I will be using clumsy in this project so that I can see how my implementations handle under certain network conditions like high packet loss/high latency or both.

4.3 Microsoft Excel

Microsoft Excel is a useful tool to tally stats and to make tables/graphs, I will be using Excel to plot my testing results so that I can nicely display them in some form of Graph.

4.4 Summary

I will be using Unity 2019.1.8f1 for this project because it's a great engine for rapid development though it doesn't have a full network solution as from writing this other than third-party network solution. So, because there is no network solution pre-built into the Unity I will be using Forge Networking as the bind to sync data across the network because it fits well into Unity and is

completely free and open sourced, this way if something isn't working I can dip into the source code and fix it myself.

5 Methodology / Development

5.1 Networking Protocol

There are two main networking protocols which are UDP and TCP. In short TCP known as Transmission Control Protocol is a protocol used in websites and reliable programs, TCP makes sure that the data sent is received using a handshake method. UDP short for User Datagram Protocol is a protocol used as an alternative from TCP where there is no handshake method which means the data has the potential to not be received on the other end so in other words UDP is unreliable and is used for when you want to send small data rapidly and not care whether it is received on the other end.

In this project I am using UDP sockets because UDP gives us control over how we handle reliability, whereas TCP handles that completely for you and some of the techniques listed below require data to be sent reliable, using TCP would cause stutters because if a packet is lost using TCP it has to resend the packet in a ordered fashioned. This way using UDP we can control all of that, including out of order packet and lost packets without stopping and having to resend packets.

5.2 Unity Integration with Forge Networking

Forge Networking is extremely easy to setup, it's nicely documented on their github page. You can get a client connected with a matter of clicks, once Forge Networking was all setup I wanted to create a generic class to represent what a networked rigidbody, for this project I've named it DynamicObject.cs, this is a centralized object which will make unity require other scripts needed to sync it over the network, to make this so that Forge knows it's a network object we have create a networked version of it using the NCW (Network contract Wizard) which comes with Forge, and linked the two.

```
[DisallowMultipleComponent]
[RequireComponent(typeof(Rigidbody))]
[RequireComponent(typeof(TransformSync))]
public class DynamicObject : DynamicObjectBehavior
{
}
```

Figure 5 - DynamicObject class template

As you can see this object now derives from DynamicObjectBehavior, this is a Forge Networking custom network object created by the Network Contract Wizard, you will also see that this component requires two other components Rigidbody and TransformSync, the rigidbody is required, well simply because that's what our objective is, to network a rigidbody simulation and the TransformSync is something we will create next, this is the component that helps to sync the rigidbody as the TransformSync implementation will also work for non rigidbodies, that way we have a bullet proof solution for networking any object in our scene with minimal effort.

The TransformSync is just a generic component which explains how this object should be synced (Should we sync the position/rotation or both or none), whether it has children game objects and how they should be synced relative to the parent. I've separated each technique, this way we can easily switch between each technique, and made them as required components attached to the TransformSync this way we can focus on one scene to do the simulation and switch between the

techniques as needed, though this isn't required if your just implementing one of these techniques. The TransformSync talk to the MasterTransformSync which in all the MasterTransformSync just keeps a list of all the TransformSyncs in the scene so that way we know which objects we should be syncing as well as how they are synced.

```
private List<TransformSync> transformSyncs = new List<TransformSync>();
```

Figure 6 - C# List of all objects we are going to network

Now that we have a base for how we should sync objects we need to hook this all up to Forge Networking. As a note I am using Binary frames to send my data, as this way you don't have the overhead of Forge Networking RPCs.

5.3 Snapshot Interpolation

Snapshot Interpolation is a technique where you have the server simulating the physics simulation and the server sends only the visual state of the simulation, and on the other side (Client) your lerp between visual states. However, when the client receives these packets you cannot just take the packet receive and apply it, there's a thing called jitter as in a GDC talk Networking for Physics Programmers (Glenn Fiedler, 2018) puts it "Jitter is time varies of packet delivery, you don't get nicely spaced inputs, you get clumps, 2 input, 1 input, 3 inputs, you need buffering".

For each packet sent to the client, it has a list of all the TransformSyncs data which is synced over the network, and the important bit is what that data might look like, in my case I sent over the network id of the object I'm syncing followed by the data which is required to interpolate for the client, for each transform we are going to sync, I send a bool flag for whether I want to sync position and rotation, this is a good bandwidth optimization because if you know you don't need to sync the rotation of an object then you can just uncheck rotation and save a bunch of bandwidth.

```
// Holds all the transforms children and the parent transform which we are going to
sync
public class TransformsSyncSnapMessage : IHawkMessage
{
    public NetworkObject networkObject;
    public HawkMessageList<TransformSnapMessage> transformsMessages = new
HawkMessageList<TransformSnapMessage>();
```

Figure 7 - A class template of the data per object

```
// Contains a single transform and what parts of the transform we are going to sync
public class TransformSnapMessage : IHawkMessage
{
    public bool bSyncPosition;
    public bool bSyncRotation;

    public Vector3 position;
    public Quaternion rotation;
```

Figure 8 - A class template of a child object

As for interpolating on the other side, I wait until I receive a packet for the object I'm going to interpolate, at this stage it only has packet A so we can't interpolate to anything however a nice little trick, is to create a fake starting packet called 'A' only on initial start as you can assume the first packet will be from initial starting position/rotation and when you receive the first packet you will have packet 'B', which is great because we can interpolate between 'A' and 'B' nicely. As for interpolating, I just using 'Vector3.Lerp' and 'Quaternion.Slerp', these give me great results.

5.3.1 Latency

Latency is the delay from which you send a communication and the time it takes for it to arrive, this is crucial for many competitive games because if data is delayed for too long then it could potentially frustrate the player or make the game feel floaty or for many Esports games, could mean an unfair advantage for other players. An article called What is network latency (and how do you use a latency calculator to calculate (Duffy, G. 2020) mentions the typical latency for a home connection would typically be between 60ms (4G) to 10ms (Ethernet)

When sending only the visual state to the client at something like 10 packets per second, you'll notice that, you'll have input delay this is non-negotiable you've got to send the input of your player to the server, and once it arrives the server will eventually send the visual state of the world packet back to you, this alone is around 200ms which is basically unplayable, also taking into account that the packet has to go through the interpolation buffer which yet alone has its own delay to ensure packets are evenly spread. I managed to handle latency by ramping up the 10 packets per second to 60 packets per second, which gives us roughly latency from 200 to around 100ms, which is still not great. What you typically find is games hide this latency, which is exactly what I did, so we locally simulate the camera movement around the player so that when you press forward it will go in the direction of the camera's forward axis at that given frame, and because we camera movement isn't tied to any network latency it feels and works nicely. You can also see this in Games like Human Fall Flat and the upcoming Wobbly Life title.

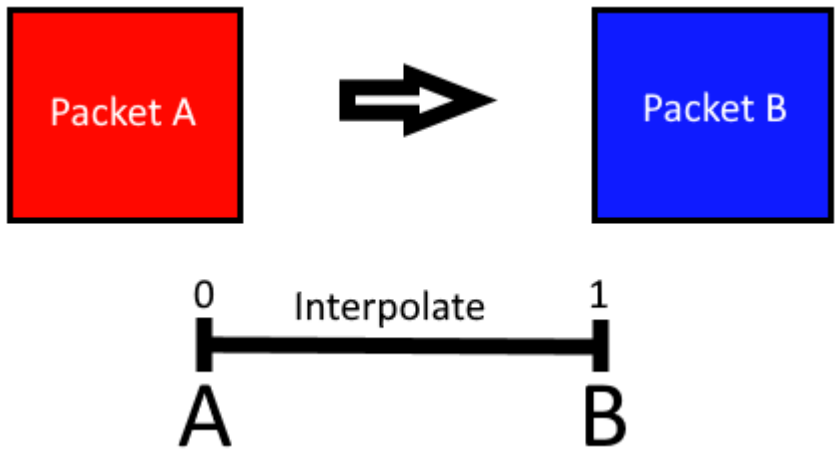


Figure 9 - Snapshot Interpolation Interpolate from A to B

5.3.2 Interpolation Buffer

In this case we will use an interpolation buffer, which will keep hold of some packets for a period before we can take it out, this way we end up with a nice evenly spaced packet. For this example, I've used a buffer of around 2 frames, this way hopefully we should always have something in the buffer. One thing to take into account is that you must be able to maintain your buffer because if you lose a packet then you'll receive a hitch and similar if you have too many packets in the interpolation buffer you increase the visual latency which basically makes the game unplayable.

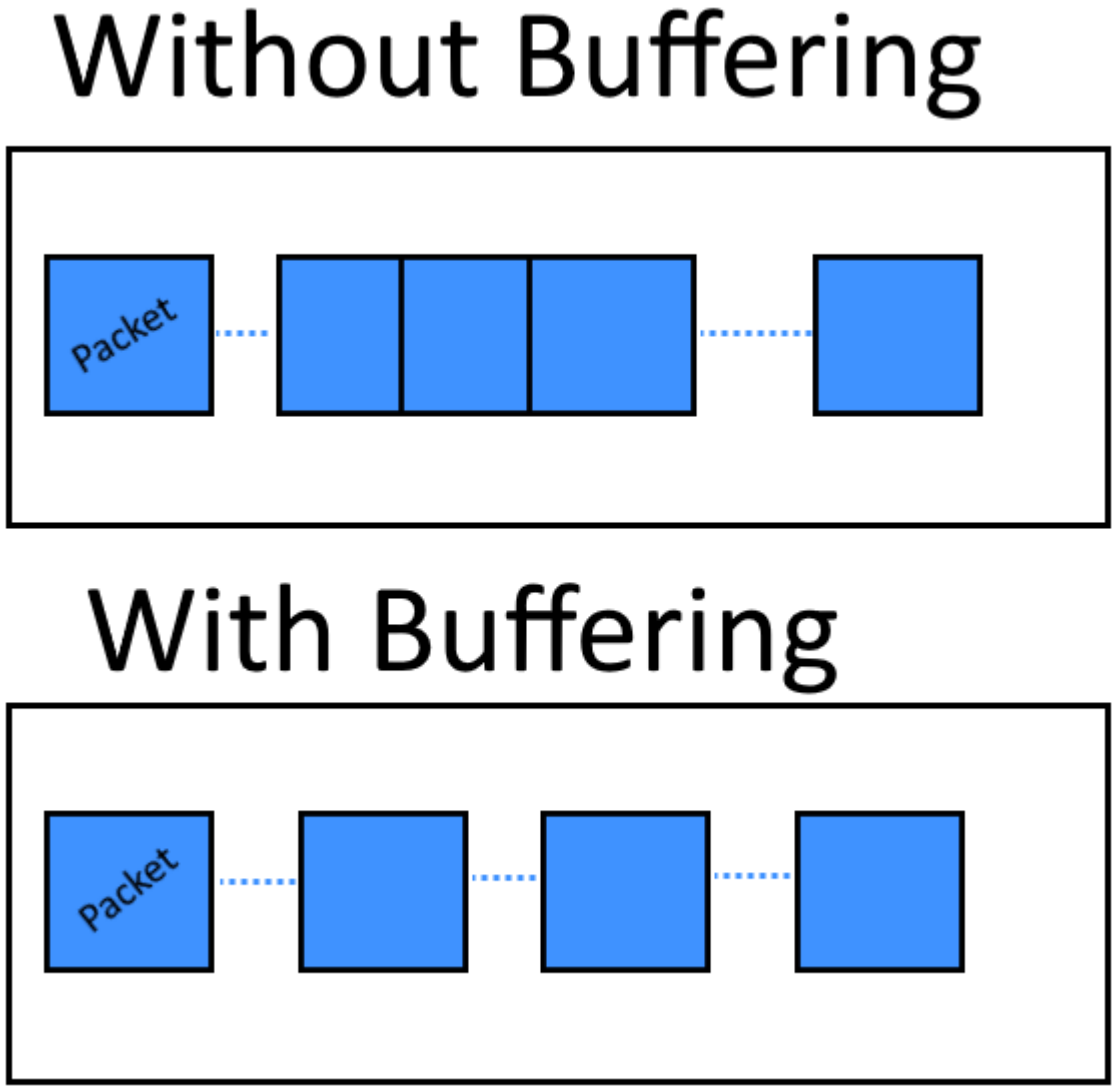


Figure 10 - Snapshot Interpolation without buffering vs with buffering

5.3.3 Maintaining the Interpolation Buffer Size

Maintaining the Interpolation Buffer is something you must do to ensure that you get evenly placed packets if you don't manage it well, you'll get hitches. In Unity 2019, I ran my interpolation buffer in the FixedUpdate loop, so that my interpolation buffer ran via ticks instead of relying on updating and timing, I updated my buffer every 1 tick at 60 ticks per second (Not default to Unity, you need to modify this in the Time settings), this way every tick the server would send a interpolation packet and at the same time the client would process one, it's rare for these to go out of sync.

At first, I let the buffer fill up, around 1 or 2 packets will do fine, as this rarely will go out of sync. This is perfect for keeping the interpolation buffer maintained however in the real world, this isn't always the case as packet loss is a thing, so what happens if a packet is lost.

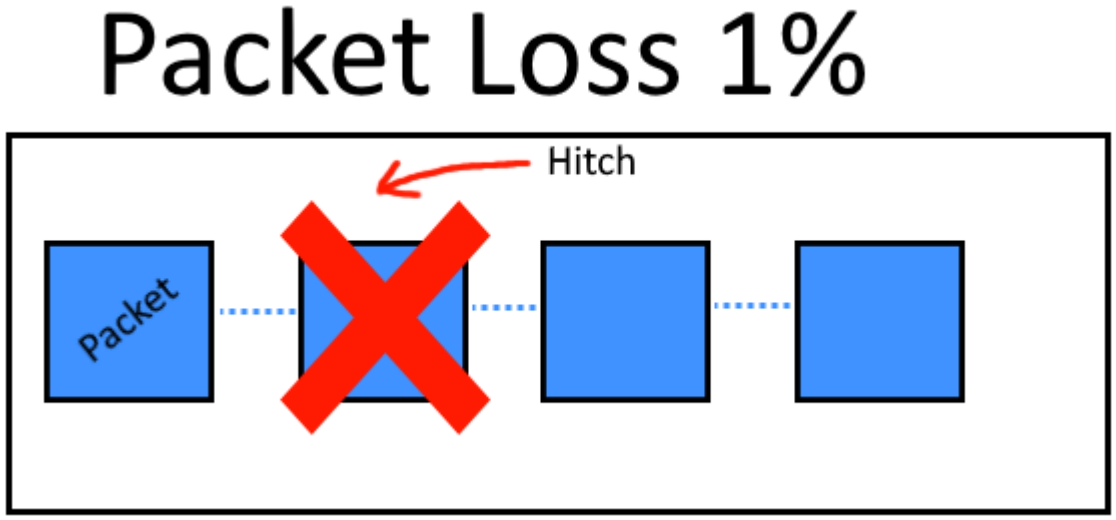


Figure 11 - Snapshot Interpolation 1% packet loss

As you can see if a packet doesn't arrive in time you receive a hitch, and there isn't much we can do about this. However from my own experimenting I managed to find a nice solution for this and that is recreating the lost packet, if we assign an id to each packet we know which order these packets should arrive in, I gave each packet a 16 bit unsigned index, so now we know the index we can detect when a packet is lost and if we receive packet with index 5 we expect to receive a packet with index 6 next and if we don't and we receive packet index 7 instead of 6, then we know we have lost packet 6 and that we need to recreate this packet, which is quite simple. As we are sending position and quaternion values, we can recreate packet 6 by interpolating from packet 5 to packet 7 by 0.5, this will give us a rough estimate of what packet 6 should look like, and from my testing this works well.

Sometimes the buffer might be too big, this is extremely rare to happen and typically will only happen if the connection is dropped for a few seconds, once it reconnects you'll get a massive lump of packets which will overfill the interpolation buffer, which in cases like this you still want to look at these packets but instead of interpolating you just snap hard, and move on until the interpolation buffer is stable again. A nice little trick I like to do is extrapolation, when there is no packets in the buffer I extrapolate the buffer from the previous packet in the buffer, this way the simulation doesn't just stop hard and instead still moves on, however because you are not simulating the physics on the client side this will look weird if it goes on too long.

5.3.4 Summary

Snapshot Interpolation is a great technique for games which require a lot of joints, as all you need to do is interpolate between snapshots, so there won't be any artefacts. Some example of games which I think utilize this technique is Human Fall Flat, Wobbly Life and possibly Totally Reliable Delivery Service, all which are active ragdoll games, which have many joints on a rigidbody, these games require the sync of their rigidbodies to be basically perfect however you can easily notice the input latency when playing these games. These games hide this latency by allowing the player to rotate their camera around their player and then sending this data to the server which then rotates the player towards the local players camera, which can also be seen in my example. However, this technique would not be great for large player counts, as you are sending large amounts of packets per second, you will easily eat up your bandwidth.

5.4 State Synchronization

State Synchronization is a technique where both the client and the server run the physics simulation, but because the physics engine is non deterministic you send the state of the object as rapidly as you can to patch up where it goes out of sync. In my simulation I send the states at around 15 packets per second, however this comes at a cost of latency, which you can implement so sort of prediction to resolve this.

One thing which is critical for this technique is when you receive the packet from the other end, you want to snap the packet hard, no interpolation just state hard, the reason for this is you are running the simulation on your side so if you start doing some interpolation then you're not running the simulation and will cause all sort of artefacts.

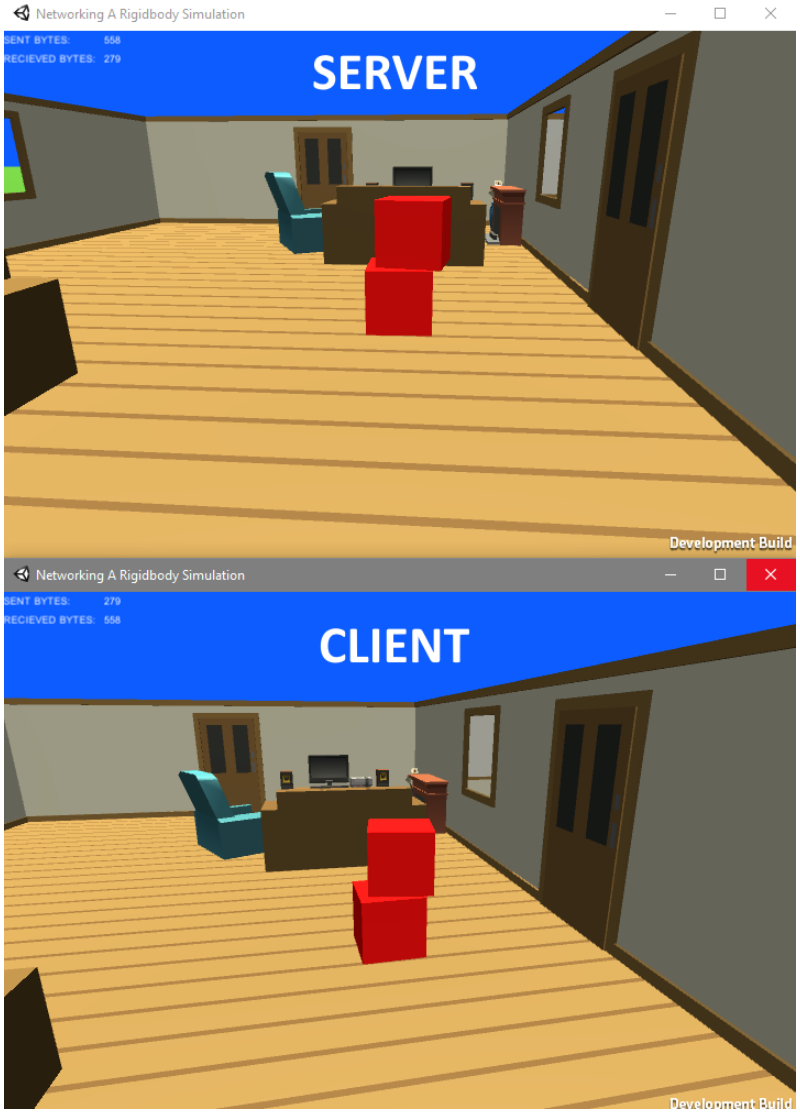


Figure 12 - State Synchronization Gameplay Image

Each packet consisted of a position, rotation, linear and angular velocity, we need to send over linear and angular velocity because we are running the simulation on both sides so it can work out the extrapolation while we are sending the next packet, I also include two Booleans for whether you are syncing the position/rotation or both, this is a good optimization because for something like a door, you only ever want to sync the hinge joint which is typically only ever the rotation, so this is a good bandwidth optimization which I found useful, however if you are sending more positions and

rotation over just sending position / rotation on their own, you might find it beneficial to not include the bools.

```
public class TransformStateMessage : IHawkMessage
{
    public bool bSyncPosition;
    public bool bSyncRotation;

    public Vector3 position;
    public Vector3 linearVelocity;

    public Quaternion rotation;
    public Vector3 angularVelocity;

}
```

Figure 13 - State Synchronization Packet definition

5.4.1 Jitter

Once again jitter exists and we need to handle it, so same as snapshot interpolation we need some sort of buffer to store incoming packets for a period of time before we can allow the game to simulate them, in my case I just used the same interpolation/jitter buffer from snapshot interpolation and used it for state synchronization. This is extremely important as if we just allowed jitter to occur it would make the game unplayable as the player will just see objects jumping from point A to point B all the time, and for games like First Person Shooters this will make the game feel unfair as players will be teleporting around even though on their screen they are walking smoothly.

5.4.2 Priority

One of the advantages of using state synchronization is that the simulation is being ran on both side so it doesn't really matter if you don't send a state for a rigidbody straight away, so one good bandwidth trick is to say don't sync any more than 20 rigidbodies and if you need to sync more than 20 rigidbodies then send the ones you didn't sync the next frame, one way you can go about doing this is by adding a priory level to the rigidbodies, in my simulation I just used a enum High, Medium and Low, High would be something like players, Medium could just be generic rigidbodies and low could be something which isn't important but does need to be synced up some point. This way you can have hundreds of rigidbodies being synced up, however the more you have the more jittery it will look on the client side.

5.4.3 Summary

State Synchronization is a great technique for when you have a bandwidth limit, this technique is probably the most utilized technique, though some example which I think use it is Grand Theft Auto 5, Red Dead Redemption 2, Halo 3 and Minecraft. This technique is great for large player count, as you can priorities your updates and limit the amount of bandwidth you can send each frame. However, it's never perfect, you will always battle to sync the simulation and sometimes it will never fully be in sync, what one player might see the other might not.

5.5 Deterministic Lockstep

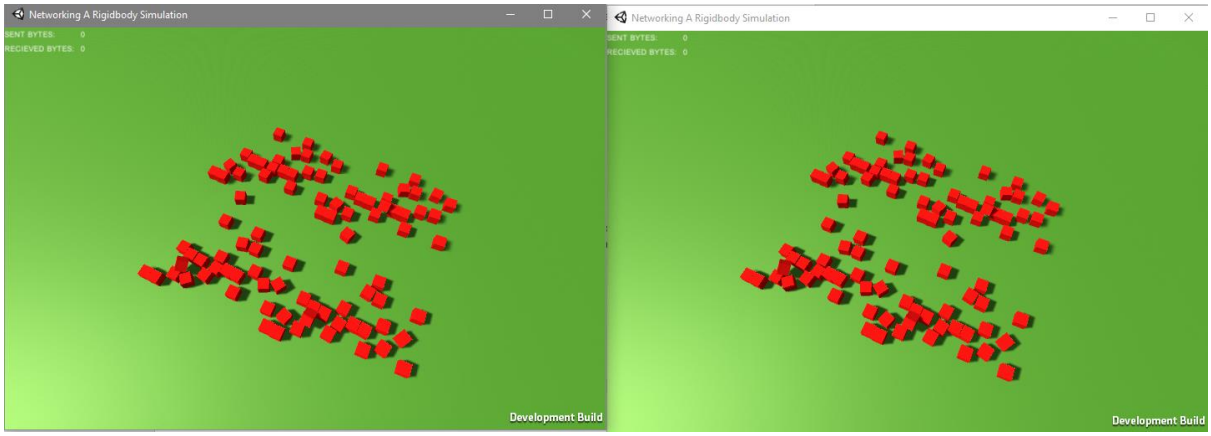
Deterministic Lockstep is another technique where unlike snapshot interpolation and state synchronisation you just send over the input required to progress on. However, this technique requires that the physics engine is fully deterministic, right down to the bit level, and because of this,

this technique typically doesn't get chosen. In my example I only ever send Player input as that's the only interacting object which requires some sort of input

```
public class PlayerDeterministicInput : DeterministicInputBase
{
    public bool bForward;
    public bool bBackward;
    public bool bRight;
    public bool bLeft;
    public bool bJump;
    public float yRotation;
}
```

Figure 14 - Deterministic Lockstep Packet Definition

I managed to get this technique to work in Unity 2019.1.8f1, as mentioned in the research chapter, there is a new update that added in Enhanced Determinism, it's supposed to make the physics engine act more deterministic as long as you always give it the same input. So, enabling that was key to getting this technique to work, however one thing to note, I have only tested this on the same PC using the same compiler, it might vary completely otherwise. However, my goal was to get deterministic lockstep to work in Unity. I also made a quick scene to test this enhanced determinism.



From the results above, this was enough indication that I managed to get Unity 2019 PhysX to run in a deterministic way, however as mentioned this is only on the same Computer, running the same OS and Compiler, you may get completely different results depending on various conditions.

In my simulation I have also added an in game UI "Press Enter to Start" this is so once everyone is in the game and loaded, the host would press enter, this is to ensure that everything is setup on all clients otherwise there could be potential issues with non-deterministic other parts of the engine and Forge Networking setting up, this is so we can rule all that out and focus on the physics engine and sending inputs only.

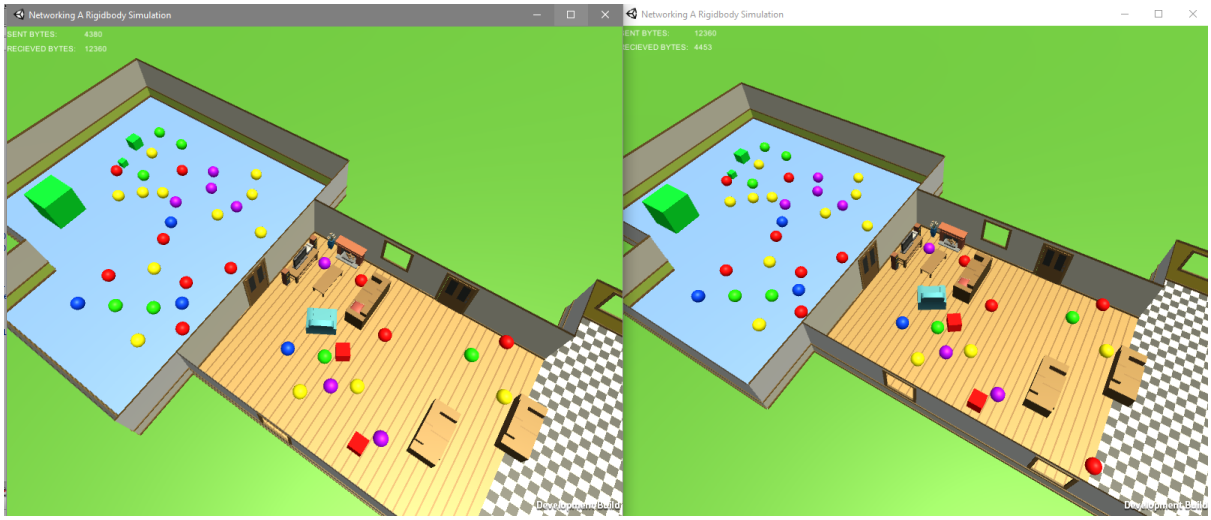


Figure 15 - Deterministic Lockstep Gameplay image

5.5.1 Jitter

Same with snapshot interpolation, jitter exists so how do we handle it. We handle it the exact same way as snapshot interpolation, in my example I reuse the interpolation buffer as it does the job well. But simply you just need to hold the input for frames so that there's always something in the buffer, otherwise the simulation will stop as there is no input in the buffer to simulate the next frame.

5.5.2 Packet Loss

Deterministic Lockstep requires all input, so what happens if we lose a packet, we simply cannot progress on until we have that input. The way I managed to get around this, is by following the same technique as an article called Deterministic Lockstep (Glenn Fiedler, 2014), which Glenn (2014) mentions to "send all inputs which haven't been acked". Which is exactly what I did, this way if an input is lost, it will arrive on the next packet, and when we simulate the input we send an ack back to the server with the input id we just simulated so that the server doesn't have to send that input again.

5.5.3 Summary

Deterministic Lockstep is another great technique for when you are on a fixed platform, as it's not guaranteed that the simulation will be deterministic on different hardware, so limiting yourself to a fixed platform like the PlayStation 4, you can hopefully ensure the simulation is deterministic. This technique is great for games which have loads of units (RTS games) or for games which are limited to a small amount of bandwidth, however determinism isn't easy and if something goes out of sync everything else will soon follow. Some games which have utilized this technique is Age of Empire and Little Big Planet 3.

5.6 Whether TCP would benefit any of the network techniques over UDP

For the three techniques listed above I do not think that using TCP would benefit any of the techniques in a real time networked game, if you are making a turn-based game then using TCP would probably benefit you. As for the reason why, I don't think TCP would benefit any of the three techniques listed above in a real time game is because even though with TCP every packet is reliable which is great it also has massive downfalls, which are if you lose a packet then the simulation will have to halt until that packet has arrived, which for a real time game you cannot have halt/stutter as the game will basically be unplayable, so using UDP and utilizing the techniques above where we are required packets to be delivered on the other end like in Deterministic Lockstep we require inputs to

be send reliable however we don't send them reliable and instead send every input which hasn't been acked by the client, which is a completely different way to handling reliability and has the benefits of not halting the simulation. Which in result gives a better experience for players.

6 Testing

Each video game has its techniques used to network a rigidbody simulation, typically they will be a variant of one of the 3 techniques listed above and carefully picked because each technique has its benefits and drawbacks. In this chapter I'll be testing my implementation of each of these techniques to see how well they perform under different simulations.

6.1 Test Plan

The test plan which I have designed will go through 3 different simulations, the first simulation which is also known as the main simulation, is a generic simulation which should work well with all three of the techniques, this simulation will contain joints and a good amount of rigidbodies. The second simulation which is also known as Joints Simulation, this simulation will be focused on joints and joints alone, this will be a good test to see how well the joints handle under network conditions. The final simulation also known as 1000 rigidbodies simulation is a simulation based on quantity of rigidbodies, this simulation contains 1000 rigidbodies not including players, this will be a good simulation to test loads of rigidbodies and whether bandwidth will be an issue for any of these techniques.

Each simulation I will be stress testing them with packet loss at around 10%, this should hopefully give a more real-world condition like environment, as this will be the worst-case scenario, and whether there will be any jitter while the simulation is active. While checking packet loss I will also be checking how much bandwidth this required for each simulation, this will be an average as depending on whether a rigidbody is active or not will depend on whether we need to send it over the network. Each test simulation will be tested with all 4 players connected to the simulation so that we can see whether the server can handle that many players as well as whether it can handle the bandwidth

6.2 Results

6.2.1 Main Simulation Test

Network Technique	Does the simulation sync without much jitter (Y/N)	Does the simulation handle packet loss well (10%) (Y/N)	Server Bandwidth Sent (Bytes Per Second)	Server Bandwidth Received (Bytes Per Second)	Client Bandwidth Sent (Bytes Per Second)	Client Bandwidth Received (Bytes Per Second)
Snapshot Interpolation	Yes, perfect	Yes perfect	87000	7100	2600	32000
State Synchronization	Yes, minor jitter	Yes for 10% packet loss the simulation handles well, but does stutter	24000	7000	2600	6000
Deterministic Lockstep	Yes, perfect	Yes, perfect	71000	11000	4380	20000

Figure 16 - Main Simulation Test Table

Snapshot Interpolation (Bandwidth/Amount of Rigidbodies)

6.2.1.1 Summary

This was an all-round simulation, for the most part all three of these network techniques handle the simulation extremely well, except for state synchronization which was always a pain to keep syncing, I fear that this will be the case for the rest of the simulation tests. Snapshot Interpolation synced and played extremely well, recreating lost packets made snapshot interpolation work well when a packet is lost. Deterministic Lockstep also was extremely successful, with 4 players it never went out of sync, however I will probably get different results if I was to use this technique using two different platforms.

6.2.2 Joints Simulation Test

Network Technique	Does the simulation sync without much jitter (Y/N)	Does the simulation handle packet loss well (10%) (Y/N)	Server Bandwidth Sent (Bytes Per Second)	Server Bandwidth Received (Bytes Per Second)	Client Bandwidth Sent (Bytes Per Second)	Client Bandwidth Received (Bytes Per Second)
Snapshot Interpolation	Yes, perfect	Yes, perfect	320000	6800	2500	103000
State Synchronization	No, the simulation is extremely jittery.	Yes for 10% packet loss the simulation handles well, but does stutter	150200	7000	2500	41000
Deterministic Lockstep	Yes, perfect	Yes, perfect	68000	11000	4400	20000

Figure 17 - Joints Simulation Test Table

6.2.2.1 Summary

This simulation focused on joints and syncing them alone, with the occasional non joint rigidbody which would be the player. Snapshot Interpolation test went extremely well, no jitter even at 10% packet loss. However, State Synchronization basically fell apart, it's kind of synced but eventually joints started jittering, which produced an unstable simulation. Deterministic lockstep simulation went extremely well, no sign of nondeterminism.

6.2.3 1000 Rigidbodies Simulation Test

Network Technique	**Does the simulation sync without much jitter (Y/N)**	**Does the simulation handle packet loss well (10%) (Y/N)**	**Server Bandwidth Sent (Bytes Per Second)**	**Server Bandwidth Received (Bytes Per Second)**	**Client Bandwidth Sent (Bytes Per Second)**	**Client Bandwidth Received (Bytes Per Second)**
Snapshot Interpolation	No, the simulation doesn't sync at all	No, the simulation falls apart	3700000	7000	2600	1000000
State Synchronization	No, the simulation does sync but there is a lot of popping	Yes for 10% packet loss the simulation handles well, but does stutter	53720	7000	2580	15509
Deterministic Lockstep	Yes, though frames do dip because of 1000 rigidbodies and running 4 instances	Yes, no stutters and continues to sync	66000	13140	4353	22822

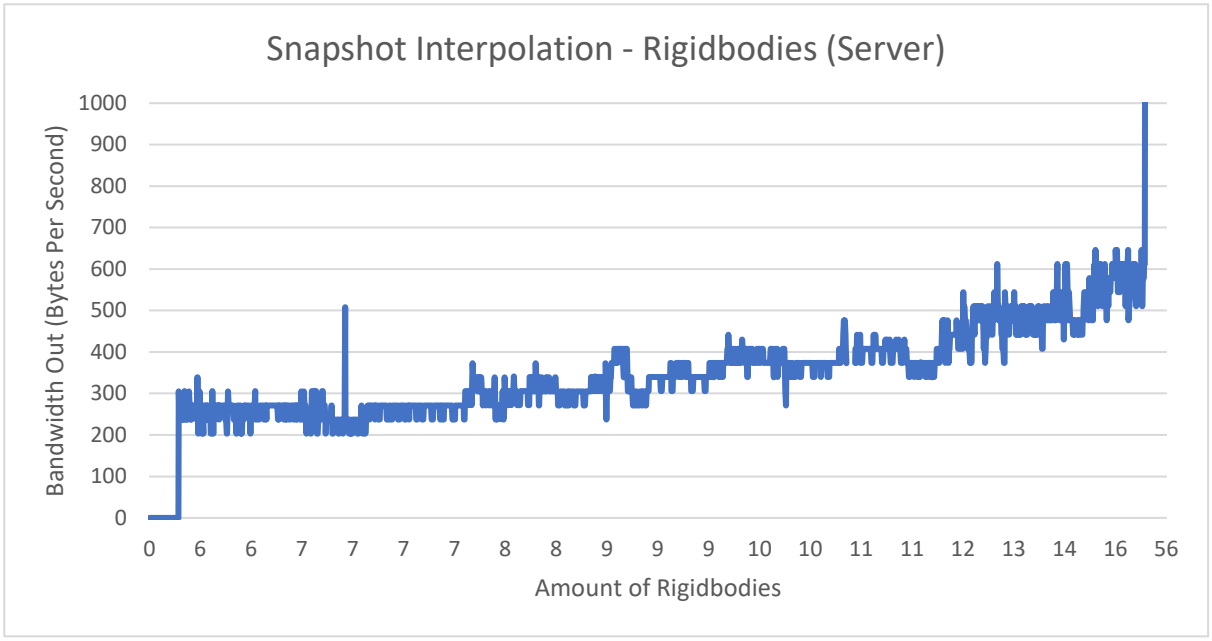
Figure 18 - 1000 Rigidbodies Simulation Test Table

6.2.3.1 Summary

This simulation focused heavily on quantity of rigidbodies, straight away snapshot interpolation fell apart as I'm using UDP you can only send a maximum of 1200 bytes per packet, forge networking does do some resizing of packets to allow for larger packet sizes which get recreated on the other end, but even with that in place the simulation just broke. State synchronization kind of synced, however there was a lot of popping, as the simulation was going through each object trying to squeeze the object into a packet and once the packet was full it would send it, that way we can ensure we sync every object at some point in the simulation, which because there was so many objects it would take some time before X object would sync causing the pop. Deterministic Lockstep worked the best out of the three techniques, with some minor frame dips as I'm simulating 1000 rigidbodies on 4 instances, which causes small frame dips, but the simulation works fantastic, to ensure it was frame dips and not the simulation I tried the simulation with two and three instances and they work much better, even under 10% packet loss.

6.2.4 Bandwidth Test (Bandwidth out on the Server)

6.2.4.1 Snapshot Interpolation



6.2.4.2

Figure 19 - Snapshot Interpolation Test Rigidbodies Quantity/Bandwidth

The above graph shows snapshot interpolation running on the server with 1 client connected, as you can see the more rigidbodies which need syncing the higher the bandwidth is required to send them. This will only get higher as there is no nice way to put a cap on the amount of rigidbodies we should send as if you can see it then we need to sync it as we don't simulate the physics on both sides.

6.2.4.3 State Synchronization

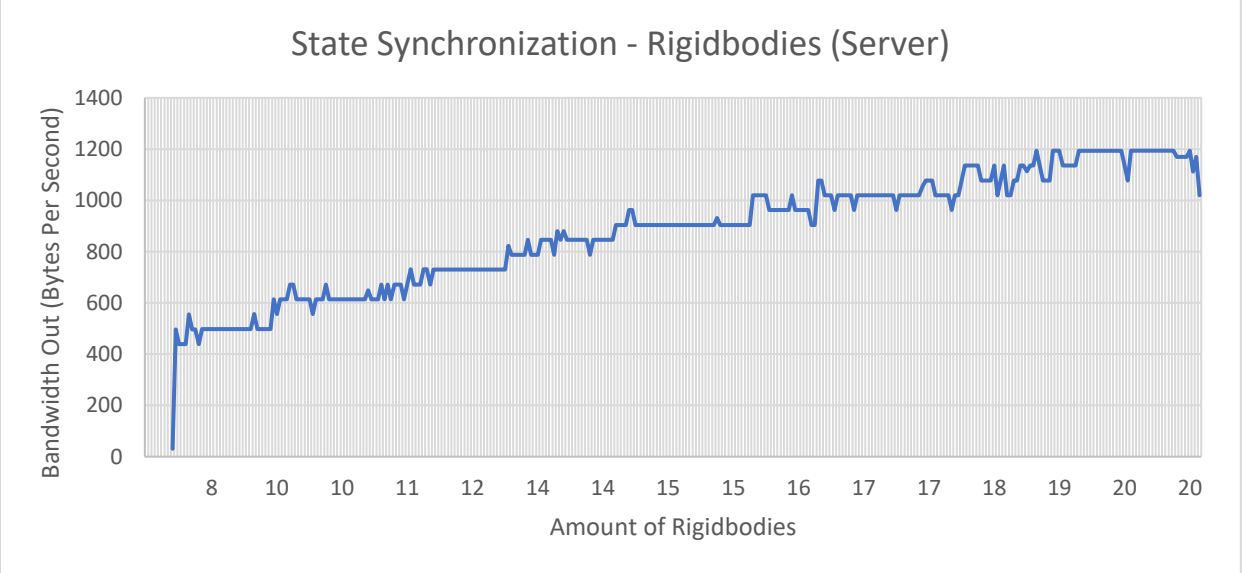


Figure 20 - State Synchronization Test Rigidbodies Quantity/Bandwidth

The above graph shows state synchronization running on the server with 1 client connected, as you can see the same with snapshot interpolation the more rigidbodies we send the higher the bandwidth, one thing to notice is that with state synchronization we only synced up to a max amount of rigidbodies per frame, which in this case is 20.

6.2.4.4 Deterministic Lockstep

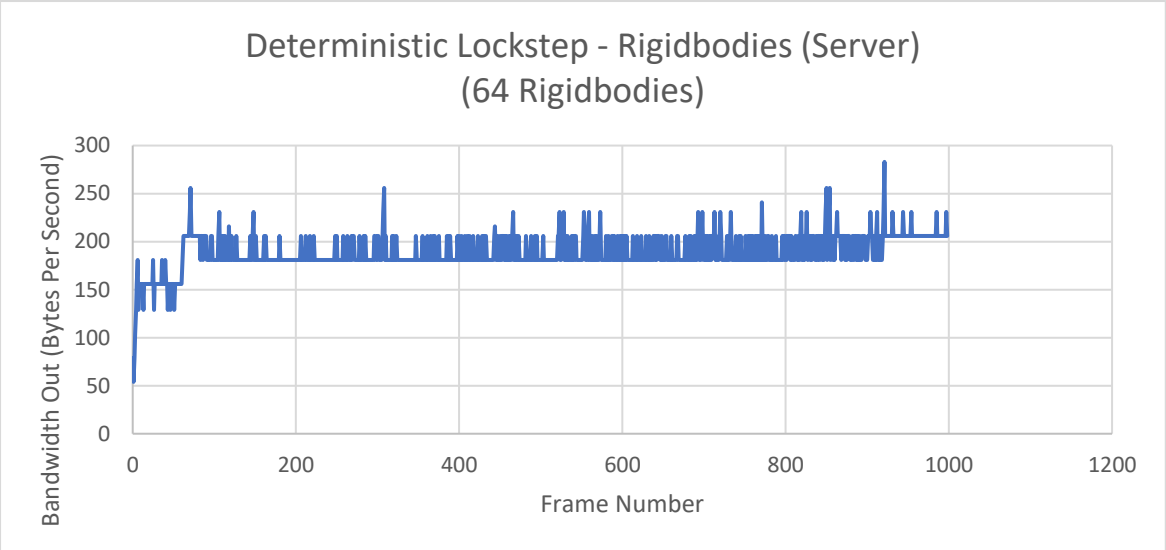


Figure 21 - Deterministic Lockstep Test Rigidbodies 64

6.2.4.5

The above graph shows deterministic lockstep running on the server with 1 client connected, the simulation is syncing 64 rigidbodies which is placed in the main simulation scene, the graph just shows how much bandwidth the server is sending to that 1 client, this is proof that deterministic lockstep is the lowest costing bandwidth technique.

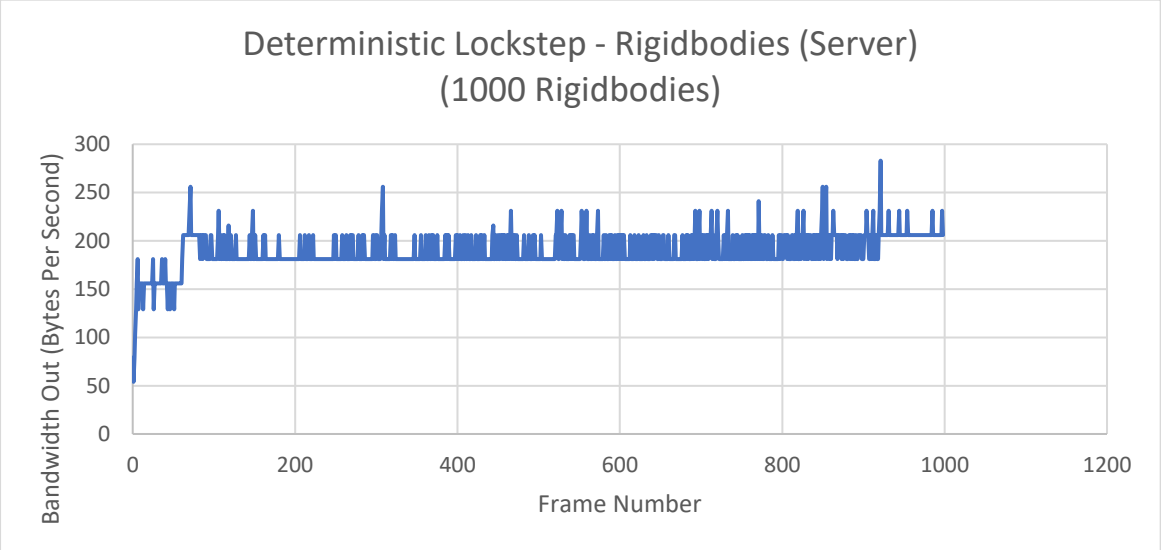


Figure 22 - Deterministic Lockstep Test Rigidbodies 1000

The above graph is another deterministic lockstep simulation which has been ran on the 1000 rigidbodies simulation scene, this graph is just proof that it doesn't matter how many rigid bodies your simulation has the bandwidth will always be roughly the same.

6.3 Analysis

After doing the 3 simulation tests on the 3 networking techniques, it's quite clear what you would use each technique for. Looking at the main simulation and the joints simulation both of these simulations work great when using Snapshot Interpolation, however when there is loads of rigidbodies like in the 1000 rigidbodies test then snapshot interpolation fails, this is because we are not running the simulation on the other side so when there is a bunch of rigidbodies we have to squeeze them all into 1 packet as we are sending 60 packets per second and because of the limitations of UDP being you cannot send a packet larger than 1200 bytes this is just a hard limit with snapshot interpolation, however snapshot interpolation is great for just syncing objects, because it's just interpolating between A and B so that's all which is required, so in your video game it is easy for you to add another object, whether that will be a vehicle, a bridge with joints whatever it is, it won't be difficult, as its just interpolating, nothing can go wrong. So, the type of game which would utilize this technique is a game which uses a lot of joints and requires the simulation to be perfect on both sides, one example is Human Fall Flat.

State synchronisation performed average out of the three techniques, it was the one technique which handled most cases, however in the joints simulation test once a few packets are lost it can make the joints jittery as they are fighting against the servers value and where they are trying to simulate towards. As the way I implemented this technique it didn't utilize any prediction as such, so input latency was an issue, however from testing this wasn't as bad as you might think. State synchronisation did perform well in the 1000 rigidbodies simulation, even though there was pops and it wasn't smooth, I think this is basically the case with state synchronisation and syncing a bunch of rigidbodies, it does work and is definitely the networking solution if your simulation isn't deterministic. Games which would typically use the technique are games are games which doesn't mind if the simulation isn't perfect time to time, but will eventually catch up, so games like Halo, Grand Theft Auto and Garry's Mod.

Deterministic lockstep performed amazing, completely unexpected, turns out you can make PhysX in Unity deterministic on the same computer. So for all the simulations deterministic lockstep overruled all the other techniques, one thing I noticed when doing the tests was that the bandwidth is extremely high for something where your just sending inputs, further investigation it seems that Forge Networking packet headers are much bigger then what I realized when choosing the network solution, so this was the reason for the high bandwidth.

7 Evaluation

7.1 Evaluation of the product

This Project was to network a rigidbody simulation, the ways you might go about doing so and potential problems which might come up when trying to sync a rigidbody simulation. I've successfully met all those objectives, I've managed to implement all three techniques in Unity 2019.1.8f1, each technique had different problems which I've solved using my own implementation.

7.1.1 Snapshot Interpolation

Snapshot Interpolation is my most solid implementation, it's always smooth and handles packet loss like a charm. However, with limitations of snapshot interpolation it didn't work well under loads of rigidbodies, so this was clear in the 1000 rigidbodies simulation. The issue with snapshot interpolation which made it quite hard to implement is the interpolation buffer, as it's quite hard to maintain this buffer as if it suddenly is empty then you'll get a hitch which doesn't look good, and this because we don't have anything to interpolate towards, I managed to solve this issue by

handling things like packet loss and recreating the lost packet, which because our buffer typically has 2 frames of interpolation we can lose a packet and we'll be fine.

Advantages

- It's easy to sync any object as you just interpolating: It doesn't matter what you are trying to sync, whether it's a bunch of joints, vehicles or just a generic rigidbody, all you've got to do is interpolate from snapshot A to snapshot B literally nothing can go wrong.

Disadvantages

- High bandwidth: As you are sending the visual state over the network you've got to send the position and rotation of every object you want to sync, this makes it very high bandwidth because the more objects you've got the more you've got to send over the network, and because we reduce latency by sending at 60 packets per second this can get very expensive.

7.1.2 State Synchronization

State Synchronization worked well enough for this project, though clearly needs a lot of improvements, one thing that I would like to add if I had more time would be some sort of client side prediction, as currently there is a lot of latency between inputs which for a game which is competitive this would not be acceptable.

Advantages

- Low bandwidth: As you are running the simulation on both sides, you don't need to send the data as rapidly as you do in snapshot interpolation, which means we can cap the bandwidth to whatever we desire, as you can use other techniques to prioritize these updates.

Disadvantages

- It's never perfect, you always adjusting the simulation: As the physics engine is not deterministic you will always keep snapping back to whatever the server values are, these can cause artefacts and jitter.

7.1.3 Deterministic Lockstep

Deterministic Lockstep worked unexpectedly, I always assumed that Nvidia PhysX is not deterministic even on the same compiler, but this project proved me wrong, I highly doubt that Nvidia PhysX is deterministic even with enhanced determinism enabled on different platforms/compilers.

Advantages

- Low bandwidth: As you are only ever sending input (which typically are small), the bandwidth will also be low
- You can have millions of objects without any more cost on bandwidth: This works because the physics engine should be deterministic, which if you are only moving the player, then you only ever need the input of the player send over the network

Disadvantages

- Limited to a small group of players: If you have any more than a small group of players typically no more then 4, you will start seeing the simulation stopping as you always have to wait for the most lagged player, as if they fall behind then the server needs to ensure they receive the inputs before carrying with the simulation

- Requires deterministic physics engine: If you don't have a deterministic physics engine then this technique will simply not work
- Determinism is hard: As well as a deterministic physics engine, you need to ensure the game logic is also deterministic, if any of this fails then the game will not be synced.

7.2 Reflection of the Development Process

The development process I think went extremely well, manging to get all three techniques working in Unity 2019 I think is an achievement. Now that I've done the project, I know the weaknesses of my implementation.

One thing that I would like to do differently next time is to change the networking solution which I used, which was Forge Networking, this was a good choice at first to get me off the ground quickly but now using it I now know it's limitations and drawbacks, which one of them is packet headers. In Forge the packet headers are much bigger then what they should be, when you send a Boolean (1 bit) you're easily add 128 bits plus more, just on the packets and that's only what I've managed to find out by digging into the source code myself. This isn't ideal so if I had more time, I would like to make my own networking solution using some sort of low-level networking library such as litenet.

Another thing which I would like to do differently next time is improve my state synchronisation by adding some sort of client-side prediction, as currently there is way too much input latency so to reduce this adding client-side prediction will resolve those input latency's .

7.3 Personal Professional Development

This project has taught me a lot, mainly a lot about networking video games and how unpredictable physics simulations really are. One thing I didn't know before this project was that Nvidia PhysX is deterministic on the same compiler/computer in Unity 2019, this was completely unexpected and I was planning on changing the physics engine so that I could implement deterministic lockstep but because Nvidia PhysX gave me the results I was looking for I didn't have to look much further.

Another thing which I learnt is how to handle packet loss when you are sending UDP packets over an unstable connection you'll get packet loss and the ways which you might go about sorting this.

7.4 Limitation of the work

One of the main limitations of my work is that I used a third party networking solution which has quite big packet headers, so when I implemented deterministic lockstep and tested it over a 4 player connection, I noticed that my data was much larger than other techniques simply because even though my data for deterministic lockstep is relatively small Forge Networking puts at least 128bits on top of my packet and when your sending 60 packets per player per frame using deterministic lockstep this easily exceeds any other technique in regards to bandwidth. Overall Forge wasn't a bad choice as it got me up to grounds quickly, which is what it's basically designed for.

Another limitation of my work is the way I implemented State Synchronisation, I feel there are better ways of handling it, preferably using some sort of client-side prediction so that there isn't much input delay

7.5 Further research / Work

As for future work, there are many improvements I would like to make. Firstly I would like to make my own networking solution on top of some sort of low level networking API like LiteNetLib instead

of using a third party high level networking API, this is so that I have full control over the data going to and from the client which will solve my packet header issue.

Another future work I would like to add to this project is adding network compression, so instead of sending Vector3 as x3 32bit float I could use techniques like delta compression to crunch this down to a much smaller value and same for Quaternions which currently are being sent as x4 32 bit floats which I believe you can easily compress into x3 32 bit floats.

Another future work I would like to add to this project is client-side prediction for techniques like State Synchronization as currently there is quite a bit of input delay which isn't acceptable for the end user.

8 References

vis2k/Mirror. *(2020)*. GitHub. *Retrieved 20 January 2020, from https://github.com/vis2k/Mirror*

What's the Better Deal Unreal Engine 4 or Unity 5?. *(2020)*. Pluralsight.com. *Retrieved 26 January 2020, from https://www.pluralsight.com/blog/film-games/whats-better-deal-unreal-engine-4-unity-5*

Sun, R. (2019). Game Networking Demystified, Part II: Deterministic. Ruoyusun.com. Retrieved 14 November 2019, from https://ruoyusun.com/2019/03/29/game-networking-2.html

Physics for Game Programmers : Networking for Physics Programmers. (2019). Gdcvault.com. Retrieved 13 November 2019, from https://www.gdcvault.com/play/1022195/Physics-for-Game-Programmers-Networking

Floating-point arithmetic. (2019). En.wikipedia.org. Retrieved 26 November 2019, from https://en.wikipedia.org/wiki/Floating-point_arithmetic

It IS Rocket Science! The Physics of 'Rocket League' Detailed. (2019). Gdcvault.com. Retrieved 13 November 2019, from https://www.gdcvault.com/play/1024972/It-IS-Rocket-Science-The

Best-Rotheray, J., & Best-Rotheray, J. (2018). Client-Side Prediction With Physics In Unity — Coder's Block - Game Development, Netcode, C++, Tea. Coder's Block. Retrieved 13 November 2019, from http://www.codersblock.org/blog/client-side-prediction-in-unity-2018

I Shot You First: Networking the Gameplay of HALO: REACH. (2019). Gdcvault.com. Retrieved 13 November 2019, from https://www.gdcvault.com/play/1014345/I-Shot-You-First-Networking

1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond. (2019). Gamasutra.com. Retrieved 17 November 2019, from https://www.gamasutra.com/view/feature/131503/1500_archers_on_a_288_network_.php?page=1

Quantum Deep Dive. (2019). Vimeo. Retrieved 13 November 2019, from https://vimeo.com/335798361/2f90c04a30

Lukas Krebs - LeadFollow Games developer (2019) Interviewed by Thomas Dunn for How he syncs rigid body simulation in Tiny Tanks.

Yakovlev, A. (2019). Physics Changes in Unity 2018.3 Beta – Unity Blog. Unity Technologies Blog. Retrieved 13 November 2019, from https://blogs.unity3d.com/2018/11/12/physics-changes-in-unity-2018-3-beta/

State Synchronization | Gaffer On Games. (2019). Gafferongames.com. Retrieved 13 November 2019, from https://gafferongames.com/post/state_synchronization/

Networked Physics in Virtual Reality: Networking a stack of cubes with Unity and PhysX | Oculus. (2019). Developer.oculus.com. Retrieved 13 November 2019, from https://developer.oculus.com/blog/networked-physics-in-virtual-reality-networking-a-stack-of-cubes-with-unity-and-physx/

Snapshot Interpolation | Gaffer On Games. (2019). Gafferongames.com. Retrieved 13 November 2019, from https://gafferongames.com/post/snapshot_interpolation/

Replicating Chaos: Vehicle Replication in Watch Dogs 2. (2019). YouTube. Retrieved 13 November 2019, from https://www.youtube.com/watch?v=_8A2gzRrWLk

Deterministic Lockstep | Gaffer On Games. (2019). Gafferongames.com. Retrieved 13 November 2019, from https://gafferongames.com/post/deterministic_lockstep/

HandmadeCon 2015 - Pat Wyatt. (2019). YouTube. Retrieved 13 November 2019, from https://www.youtube.com/watch?v=1faaOrtHJ-A

Pusch, R. (2019). Explaining how fighting games use delay-based and rollback netcode. Ars Technica. Retrieved 13 November 2019, from https://arstechnica.com/gaming/2019/10/explaining-how-fighting-games-use-delay-based-and-rollback-netcode/

'Overwatch' Gameplay Architecture and Netcode. (2019). Gdcvault.com. Retrieved 13 November 2019, from https://www.gdcvault.com/play/1024001/-Overwatch-Gameplay-Architecture-and

Making a Multiplayer FPS in Unity - YouTube. (2019). YouTube. Retrieved 13 November 2019, from https://www.youtube.com/playlist?list=PLPV2KyIb3jR5PhGqsO7G4PsbEC_Al-kPZ

Unity-Technologies/FPSSample. (2019). GitHub. Retrieved 13 November 2019, from https://github.com/Unity-Technologies/FPSSample

Deep dive into networking for Unity's FPS Sample game - Unite LA. (2019). YouTube. Retrieved 13 November 2019, from https://www.youtube.com/watch?v=k6JTaFE7SYI

Introduction to the DOTS Sample and the NetCode that drives it - Unite Copenhagen. (2019). YouTube. Retrieved 13 November 2019, from https://www.youtube.com/watch?v=P_-FoJuaYOI

"Battle(non)sense", C. (2017). How netcode works, and what makes 'good' netcode. pcgamer. Retrieved 13 November 2019, from https://www.pcgamer.com/uk/netcode-explained/

Ronacher, A. (2019). Networking: How a Shooter Shoots. Kotaku. Retrieved 13 November 2019, from https://kotaku.com/networking-how-a-shooter-shoots-5869564

non-determinism?, H., & Pflughoeft, B. (2011). How are deterministic games possible in the face of floating-point non-determinism?. Game Development Stack Exchange. Retrieved 17 November 2019, from https://gamedev.stackexchange.com/questions/14776/how-are-deterministic-games-possible-in-the-face-of-floating-point-non-determini

BeardedManStudios/ForgeNetworkingRemastered. *(2020)*. GitHub. *Retrieved 21 February 2020, from https://github.com/BeardedManStudios/ForgeNetworkingRemastered*

RevenantX/LiteNetLib. *(2020)*. GitHub. *Retrieved 21 February 2020, from https://github.com/RevenantX/LiteNetLib*

PUN vs. Bolt | Photon Engine . *(2020)*. Doc.photonengine.com. *Retrieved 21 February 2020, from https://doc.photonengine.com/en-us/pun/current/reference/pun-vs-bolt*

Unity Multiplayer - What are the pros and cons of available network solutions/assets. *(2020)*. Unity Forum. *Retrieved 21 February 2020, from https://forum.unity.com/threads/what-are-the-pros-and-cons-of-available-network-solutions-assets.609088/*

Epic Games | Store. *(2020)*. Epicgames.com. *Retrieved 17 March 2020, from https://www.epicgames.com/store/en-US/about*

Duffy, G. (2020). What is network latency (and how do you use a latency calculator to calculate throughput)?. Sas.co.uk. *Retrieved 17 March 2020, from* *https://www.sas.co.uk/blog/what-is-network-latency-how-do-you-use-a-latency-calculator-to-calculate-throughput*