Jimmy Meder

Netcode Improvements for a Mobile First-Person Shooter Game

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and communications technology

Bachelor’s Thesis

1 May 2023

Abstract

Author:	Jimmy Meder
Title:	Netcode improvements for a mobile first-person shooter game
Number of Pages:	55 pages
Date:	1 May 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information and communications technology
Professional Major:	Game Development
Supervisors:	Antti Laiho, Senior Lecturer

This thesis details on the work done to improve netcode in a first-person shooter game for mobile platforms, while being employed as a Junior Programmer for the owners of the game. These netcode improvements updated the way the game handled gameplay related messages between client and server. These changes include increasing the rate at which the server sends data to the clients, adding interpolation buffering to the client, implementing a jitter buffer on the server, synchronising clocks between client and server, and collecting performance metrics.

This was done to rid the game of a long-standing issue with stuttering movement. The improvements started out as improving character movement interpolation with the aid of buffering. This alone did get the results expected, so many changes were introduced to support this.

The documented changes improved character movement and fixed an issue with peeker's advantage, where enemies would appear into view out from behind corners, but in doing so introduced more delay between clients and server, in turn adding a different issue with peeker's advantage.

Releasing the changes into a playable beta for the players received positive feedback, but due to the number of these changes, further monitoring and adjusting needs to be done.

Keywords: netcode, network, jitter, buffer, interpolation, update, movement, stutter, fixed update, time synchronisation, time drift

Tiivistelmä

Tekijä:	Jimmy Meder
Otsikko:	Nettikoodiparannukset ensimmäisen persoonan ammuntapeliin mobiililaitteille
Sivumäärä:	55 sivua
Aika:	1.5.2023

Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaaja:	Lehtori Antti Laiho

Insinöörityön tarkoituksena oli parantaa ensimmäisen persoonan ammuntapelin näkyvien hahmojen liikkeen sulavuutta työn tilanneelle yritykselle. Nämä muutokset paransivat kyseisen pelin pelattavuuteen liittyvien verkkoliikenne-viestien käsittelyä. Toteutus alkoi yksinkertaisena, mutta kasvoi ajan myötä erittäin suureksi työksi. Se sisälsi palvelimen viestien lähetystiheyden nostamisen, jitter-puskurin toteuttamisen palvelimeen, palvelimen ja käyttäjän välisten kellojen synkronoinnin sekä suorituskykymittaustulosten keräyksen.

Muutosten tavoitteena oli korjata pelin pitkäaikainen epätasaisen liikkeen ongelma. Kehitys alkoi hahmojen interpolaatioparannuksista. Tämä yksinään ei kuitenkaan tuottanut toivottuja tuloksia, joten monia muita muutoksia lisättiin.

Dokumentoidut muutokset paransivat hahmojen liikettä ja korjasivat ongelman ”peeker’s advantage”-ilmiössä, jossa viholliset saattoivat ilmestyä näkyviin kulmien takaa. Näiden muutosten mukana peliin lisättiin palvelimen ja käyttäjien välille suurempi viive, joka toi mukanaan eri ongelman ”peeker’s advantage”-ilmiöön.

Toteutuksen lopussa näillä dokumentoiduilla muutoksilla varustettu pelattava pelin beetaversio julkaistiin käyttäjille testattavaksi. Tämä pelin kokeellinen versio sai positiivista palautetta testin aikana, mutta muutosten määrän vuoksi peli vaatii jatkuvaa tarkkailua ja mukauttamista tulevaisuudessa.

Avainsanat:	nettikoodi, jitter, puskuri, interpolaatio, liike, kello synkronointi

Contents

List of Abbreviations

FPS:	First-Person Shooter. A game that is displayed through the eyes of the playable character. Usually involves the action of shooting but is not always required even though the name suggests so.
P2P:	Peer-to-Peer. A network architecture that shares the workload between all peers with equal responsibilities. This architecture is used often in video games in place of hosting servers.
VoIP:	Voice over Internet Protocol. A system that uses the internet to transmit voice and other multimedia content

1 Introduction

Playing games is often viewed as a solitary experience, and in many cases that is the best way to enjoy them, alone. Despite those preconceptions, many games do have a more social aspect to them in the form of multiplayer, either cooperative or competitive. Whether a game is experienced by a single player, or by multiple players with harsh competition over an objective or a single common goal, games have a wondrous effect of bringing people together. Be it by just talking to friends about great single-player games and experiences, winding down and dissecting an intense competitive match over smiles and laughter, or experiencing a game together cooperatively, the games bring people closer together.

This thesis focuses on games played over a networked connection with the focus on netcode improvements made for a game called Critical Ops, a game by Critical Force. The work and research were done while working as a Junior Programmer for Critical Force. Development was done in small teams in Critical Force; thus, all the work was done by Jimmy Meder, a Junior Programmer at Critical Force, and Nathan Biagini, a Network Developer at Critical Force.

Netcode is a term used for how a game handles communication and synchronization between players and servers during gameplay. Given how competitive the online gaming market is, the rate at which games evolve, and how hardware used for games advances, a game’s netcode needs constant improvements, or players will soon find it inadequate.

2 Background Information

Critical Ops is a First-Person Shooter (FPS) game for mobile platforms made by Critical Force. Critical Ops runs with the Unity game engine. The company behind Critical Ops, Critical Force, was founded in 2012 by Veli-Pekka Piirainen along with a small group of game development students. It got its origins from KAMK, Kajaani University of Applied Sciences, where founder and former CEO Veli-Pekka Piirainen worked teaching about game development. From there, Critical Strike Portable that started as a hobby project, soon grew out to become the precursor to Critical Ops. [1,2.]

Critical Strike Portable, also known as Critical Missions: SWAT, was based on the popular title Counter-Strike. It was originally released as a WEB game on December 26th in 2011. The game got a good amount of popularity and was then further developed on mobile. This gathered a lot of attention, as there were not many first-person shooter (FPS) games on the mobile market at the time. Critical Strike Portable was released on mobile on January 2nd in 2013. Development for Critical Strike Portable would later be discontinued in 2014. [1,3,4.]

The company would start development for Critical Ops in 2014. The game would be a successor to Critical Strike Portable. The goal of this new project was to learn from the pitfalls of Critical Strike Portable and bring a high-quality 3D FPS game for mobile devices. The game has been live since 2015 and is still going strong to this day. [1,5.]

Critical Ops' movement-related issues can be dated back to 2019 when they were first noticed by players. By 2020, there were different views on the cause of these issues, but nothing was clear. Understanding the issue from the players' feedback would prove difficult. The problem would appear to the players as teleporting movement and in other ways that could be attributed to a bad connection, which led to investigating in the wrong places. Between 2020 and 2022, attempts to remedy the issue were made. However, during this time, other issues came up and got fixed. For the average player, this did not affect playability much, but to the very active core players in an Esports game, it was very distracting. All of this affected the balance of the game and had to be compensated for, affecting game design as well. [6.]

Many things that are out of the control of the player and developer affect how a player might experience an online game. A player might have a bad connection and experience increased network delay (also referred to as lag) or network packets may get lost in transit. These and a few more factors can cause the game in question to feel subpar to play. In Critical Ops, an issue was noticed even with good network conditions, that remote players’ characters (players other than the person holding the device) would stutter while moving, slowing down and speeding up at random intervals and teleporting short distances. These were very small but very frequent, causing the remote characters to move in an unsatisfactory way, and as a side effect, making hitting enemies a little harder. Though the issue was not new, but dated back years, this was still unacceptable and needed to be addressed.

The goal was to get remote characters to move in a much smoother and more predictable way. To achieve this goal, Nathan Biagini introduced an interpolation buffer (a concept that will be expanded upon later) to the game. In short, what this did was delay the processing of network data on each client to absorb some of the randomness network conditions may cause. A more thorough explanation of this concept is expanded upon later in this thesis. While the interpolation buffer worked, it did not completely fix the issue on its own. Other ways to normalize network information processing needed to be introduced.

3 Online First-Person Shooter Games

3.1 Evolution of the Online Shooter Game

Online gaming is an ever-growing pastime with many different genres of games having a secure place in that market, with FPS games being a long-time staple in that scene. FPS games have been a driving force in development for the online gaming experience, with its roots dating back to December 10th in 1993, with the release of Doom. This revolutionary FPS game had the possibility to play with up to three other players, for a four-player deathmatch game. [7,8.]

The online experience of Doom was limited to a local network, which severely restrained its popularity and availability, as many did not have access to multiple devices on a single local network to play with. [8]

In 1996, Id Software released Quake, a game that would popularise online gaming and would serve as the base for many developments in online gaming [9]. The game supported both competitive and co-operative online gameplay. The original Quake worked with peer-to-peer (P2P) architecture and was released with netcode that did not perform well with poor network conditions. It did, however, work well in a Local Area Network (LAN), which gave rise to the popularity of LAN parties. These LAN parties would turn out to be crucial for the early development of Esports into what we know them today. The groups of friends made in these small gatherings would ultimately venture out and test their mettle against other groups of friends. Soon these groups would be known as clans, and the meetings would grow in size. This eventually led to competing against each other for a prize in tournaments. [9,10.]

Not satisfied with the issues Quake had with online gameplay, John Carmack, the lead programmer of Quake, started implementing several systems that would increase the quality of online play by a huge margin. This update would be called QuakeWorld. These systems would include having a server authoritative client-server architecture, clients having persistent accounts to play with, and client-side prediction to deal with the increased input delay. While not all these ideas were originally concocted by John Carmack, Quake did however popularise them. Many of these technologies are still widely in use to this day in games of all magnitudes. [11.]

Despite the many positives, QuakeWorld brought many changes that did not sit well with everybody. The changes opened Quake up to a much wider audience, where everyone with almost any quality of connection could enjoy the online play. The changes in physics and increased delays made some users, who preferred the responsiveness of LAN play, upset. This made it clear that no one solution is the best, especially when it comes to online gaming.

On November 19th in 1998, a game running a modified version of the Quake engine, GoldSrc, would yet again revolutionize the video game industry. The game was Half-Life by Valve Corporation [12,13]. The game engine that ran Quake would be used and modified by many developers. Both the original Quake engine and many of its modified iterations are still in use on the day of the writing of this thesis.

In 1999, development for a Half-Life mod called Counter-Strike was started by developers Minh "Gooseman" Le and Jess Cliffe. The popularity it quickly gathered got the attention of Valve Corporation, who then quickly bought the intellectual property and hired Le and Cliffe to continue work on the game. The first non-beta version of the game was released on November 9th in 2000 [14,15]. The game had such a profound influence on the online gaming scene, that it quickly became the standard for online gaming. Counter-Strike was responsible for shaping the Esports scene heavily, and at the time of writing this thesis, Counter-Strike still is one of the most recognizable titles in Esports, with large tournaments hosted annually.

Since then, many ideas and developments, some small and some large, have been made to improve the online gaming experience. This after all will likely never be perfected, as there are many different solutions for many different situations, and no one solution is best for all.

3.2 Netcode Authority

In online games, the game must send and receive data from others to update their end on the actions of other players. This can be divided into two categories, Peer-to-peer, and client-server architecture, with each having its pros and cons. Both architectures can be further divided into subcategories. P2P architecture is still used today, while client-server architecture is the more popular one of the two.

Peer-to-peer

Peer-to-peer architecture has clients passing data on the game states directly to each other [16]. This architecture can be divided into two subcategories. The first subcategory has a designated host through which all states go through, making this more akin to a client-server architecture. The second subcategory is when there is no single host, but all clients send states directly to every single other client connected to the game session; this type of architecture was used a lot in older games.

P2P gaming has some issues in both subcategories. Connecting clients directly to each other means that every client has their IP address exposed to every client connected to the same game session to be able to directly communicate with them, which raises issues about security. Having no central monitored authority in this architecture means that cheating in-game is much easier and punishment for causing trouble is much harder to uphold. [16.] Network delay (lag) caused by long-distance connections is much higher and harder to compensate for on the netcode side.

Not all was bad with P2P architecture. Having the responsibility of hosting and maintaining online games shift to the players is a major saving in both costs and complexity [16]. Likewise, if P2P connections had little network delay, such as in a local area network, the experience for the players could often be superior as almost no input delay is introduced through the netcode. In some situations, a

P2P architecture for an online game can be preferable, such as in older games, indie games, and local area network gaming.

Client-server

Client-server architecture has a central designated machine working as a server to which all clients connect, and each client only communicates with the server. With dedicated servers handling online gaming, the responsibility of hosting games and maintaining servers is shifted to the developers, increasing cost and complexity. [17.]

Client-server architecture can improve the network conditions of clients by having servers guaranteed to be capable of processing the game server and the network traffic, and by having a network of regional servers guaranteeing each client can connect to a server only a short distance away. Two guarantees that client hosting cannot vouch for. [17.]

A centrally controlled server that all clients connect to for each game session also improves security by only exposing the IP address of the server for each client. This central server also makes cheating much harder, as all network messages sent by each client go through a single monitored point, giving the developers the possibility to detect tampered network messages. These factors contribute to client-server architecture being more popular than P2P when it comes to larger game developers, who have the resources to run a network of servers. [17.]

Looking a bit deeper into client-server architecture, the presence of a central server that all network messages go through does not necessarily mean that it has authority over what happens during a game. If the client sends the server information on how it interacted with the world, what its current position is, how fast they are moving, what they are currently doing, etc., and the server accepts these and updates the server’s game world state accordingly, then sends this updated state to all other clients, the game is considered client authoritative.

Games that are fully client authoritative perform no validation on the server. [18]. If a game server accepts all states any client sends them, even those that are tampered with, the issue of cheating becomes very relevant.

When the messages sent to the server are merely requests and all actions are performed by the server, the game is then called server authoritative. Games that are fully server authoritative have their servers receive only input prompts from the client, perform all relevant actions in the server, and send updated states of the game back to the clients. This makes sure nothing impossible is performed in-game.

In a server-authoritative architecture, when a client sends an input over the network to the server, they must wait for the server to process it and send an updated state back, introducing a large delay to the game between input on the controls and output on the screen. This delay can make the game feel unresponsive and in a fast-paced shooter game, this is unacceptable.

A game does not have to be absolute in its implementation of either client-authoritative or server-authoritative architecture. Instead, varying degrees of both can be implemented to combat cheating and improve the feel of the game. For example, the server may trust the client's messages on actions that do not affect gameplay, validate other more gameplay-affecting messages, and accept only inputs for gameplay critical events. There are other ways to improve the feel of fully server-authoritative games, such as client-side prediction and lag compensation [19], but those are not the focus of this thesis and a plethora of amazing material regarding those subjects can be found easily. However, understanding different game server architectures, their pros and cons, and the issues each different architecture has, is important in understanding the netcode and network solutions in Critical Ops and why certain solutions were used.

3.3 Peeker’s Advantage

Peeker’s advantage is a phenomenon caused by slight network desynchronization between players. The in-game effects of which can be seen in figure 1. To explain peeker's advantage, let's build a scene. Player A is holding position at a point at the end of a hallway, looking at the entrance to said hallway. Player B is approaching Player A from behind the corner at the hallway’s entrance. Player B rounds the corner into the hallway, sees Player A, and takes a shot, taking a total of 0.6 seconds. Player A sees Player B come around the corner and takes a shot, taking a total of 0.4 seconds, but Player A still gets eliminated. Player A reacted faster but still lost the exchange due to peeker’s advantage. Player B sees Player A as soon as their device has processed the inputs. Player A sees Player B after actions are received and processed on the server, then new states are received and processed on Player A’s device. [20.]

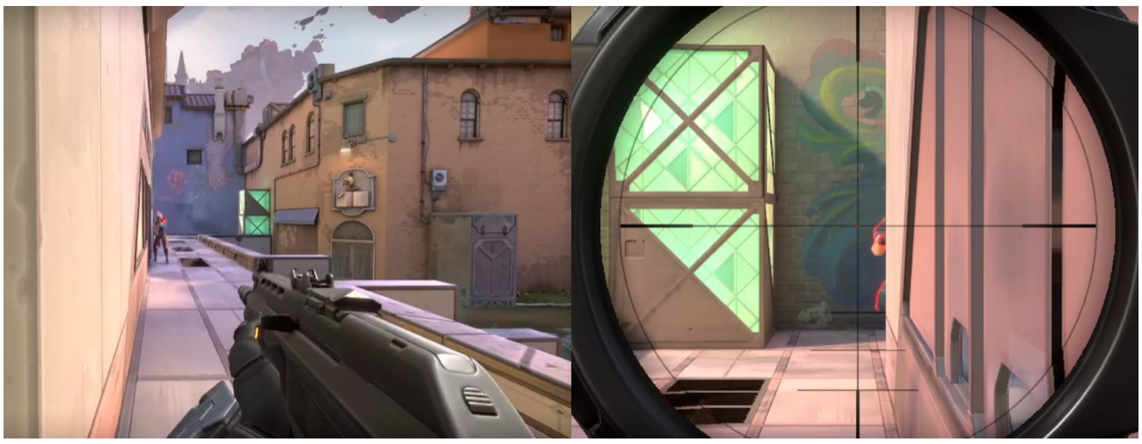


Figure 1 Peeking a corner in Valorant from both perspectives. [20.]

Peeker’s advantage is not caused only by the network. The way messages are handled on clients and servers can also amplify this phenomenon. The effects of buffering and processing delays are demonstrated in figure 2, where the sphere on the left represents the position displayed to the players, and the sphere on the right represents the actual last received position from the server.

In the case of this example, the line in figure 2 equates to 150 milliseconds of buffering, or 0.15 seconds.

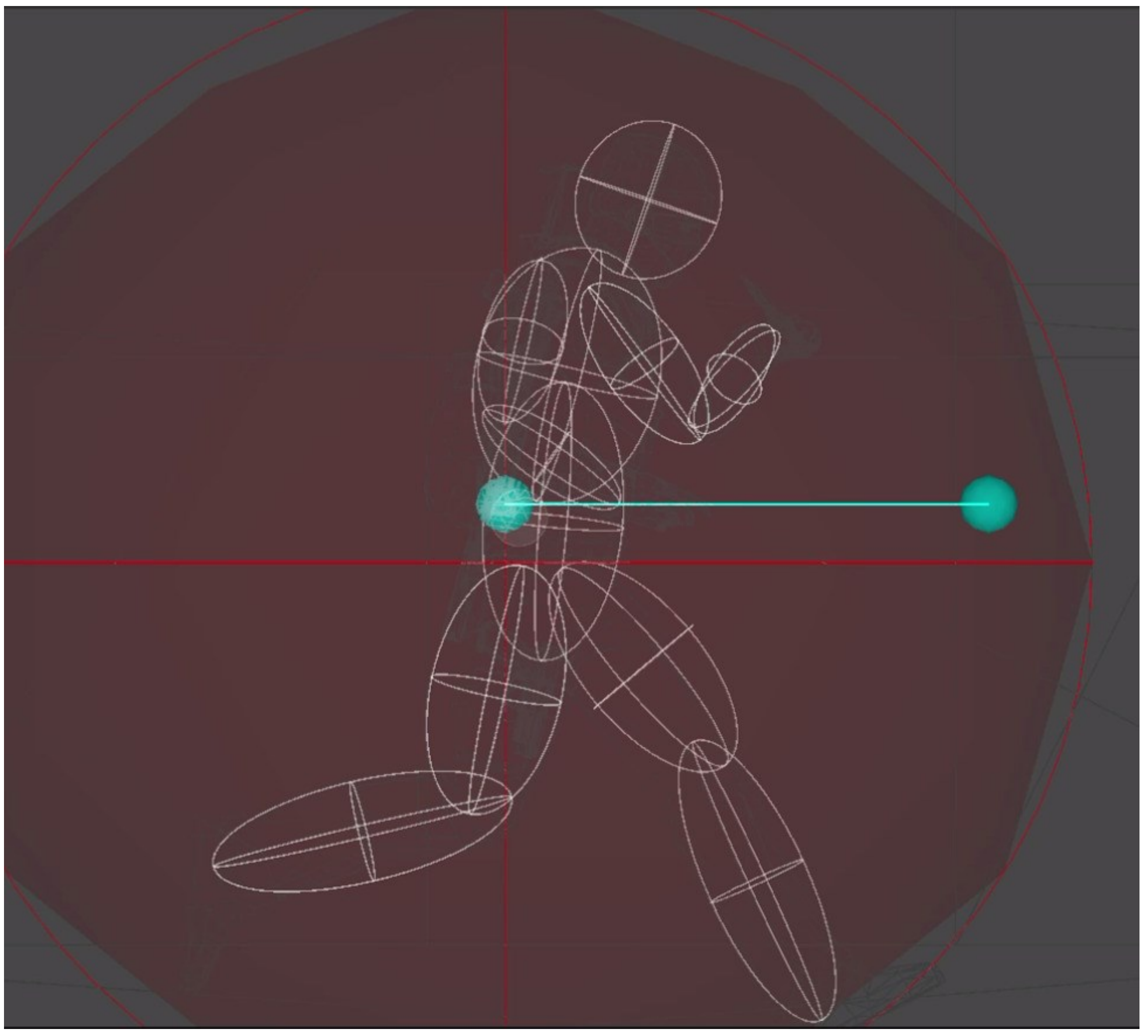


Figure 2 Difference between current and displayed positions caused by buffering.

From a practical point of view for a local client in relation to the server, the local client is “in the future”, the server shows the local client the “present state” and every remote client is “in the past”. From a technical point of view, the delays caused by processing and network variables cumulate to a visible desynchronization between remote and local client states. What this means is that what a player sees is actually more than 150 milliseconds in the past. With the average human response time to visual stimulus being around 250 milliseconds [21], the holder may not have time to react to a character moving from behind a corner, giving the peeker an advantage.

Most of the time the developers have little to no influence over network delays other than having more servers closer to each other. However, there are many ways the developers can affect peeker's advantage in the game's way of processing, from minimizing the delay between receiving and handling of inputs to visual tricks that help to negate the feeling of unfairness. Yet, the fact of the matter is that minimizing peeker's advantage is not so simple. Often delays in processing need to be introduced to negate network-induced variables and refresh rates need to consider device limitations.

3.4 Network Variables

If a game communicating with a server had no delays, all messages were received at precise even intervals, and all messages were present and accounted for, there would be no need for any of the work documented in this thesis. Reality however is never that simple. Networks are never completely reliable and need systems to keep communication over a network functional.

Network Delays

Network delay, also called end-to-end delay or "lag", is the time it takes for a sent message to reach its destination over a network. This delay is comprised of four different parts, transmission, propagation, queuing, and processing delays. [22.]

Transmission delay is the time it takes for the full message to be received on the other end. This is affected by how much data can travel through the network. For video games, this is usually not a limiting factor, as the messages are often small enough to be sent and received all at once. This delay can be observed when downloading large files, such as updates for a game. [22.]

Propagation delay is the time it takes for data to travel from end to end. This is affected by the distance the data travels. This delay has a heavy effect depending on the type of the game. Propagation delay does not carry much weight if the decisions made in a game have no time restriction. On the other hand, having frequent real-time decisions being delayed traveling over a network makes for a poor experience. Not much can be done by game developers to affect this delay, other than making sure the players have access to servers closer to them. This delay can be observed when connecting in rural areas. [22.]

Queuing delay is the time the packet spends waiting for processing at any point. This is affected by the amount of network traffic. If a point in the network is busy processing a large load of messages, further messages will be queued to wait for their turn. This delay cannot be influenced much by developers. This can be observed when connecting somewhere under heavy load. This can be artificially caused by some through a denial-of-service attack. [22,23.]

Processing delay is the time it takes for the receiver to process the received message. This, like propagation delay, can have a heavy impact on a game, but unlike other delays, the developers have much influence over this. Most of the time, some processing delay needs to be added to compensate for other variables. This delay is difficult to observe as this delay is not revealed to the user.

Network Jitter

Jitter is the variation in time in network delay. Messages over a network take different times to reach their target. A message can take a different route to its destination, even though the sender and receiver stay the same between messages. A message can run into a busy part of a network and be delayed further by congestion. Jitter causes packets to be received at uneven intervals, and at worst can cause messages to be received out of order. This variable is impossible to predict, yet it remains crucial to compensate for in many cases. [24.]

Packet Loss

Packet loss, or packet drop is when a sent message never reaches its destination. There are different protocols to ensure that a message is always received, such as by requiring verification from the receiver. [25.]

In some cases where an individual message is not critical to the function of the game, there is no need to ensure that each message is received. In this situation, these lost packets within reasonable limits can be remedied through code.

Lost packets cannot be predicted, and the consequences of a lost packet can be disastrous. The effects of packet loss can, fortunately, be prevented, as long as not too many packets are lost.

3.5 Netcode in Critical Ops

Critical Ops uses a client-server architecture and is mostly client authoritative, as explained in Chapter 3.2. To prevent blatant cheating, Critical Ops' servers use validation to check for impossible changes between the current state and the previous state of the client. To compensate for varying processing speeds for states, a fixed update is used to normalize the update interval to every 30 milliseconds, which translates to updating the server state ~30 times per second. This is referred to as a “tick rate” of ~30Hz. Fixed update will have a more detailed explanation in Chapter 4.3.

Critical Ops has opted for a mostly client-authoritative architecture. The server checks every gameplay state received from each client to prevent cheating and sends out corrections for states that have odd values.

4 Netcode Improvement Extents

4.1 Current Implementation

Critical Ops had a known issue with remote characters, where their movement was visibly irregular. Though the issue did not make the game unplayable and was small, it was hard to ignore if noticed. Characters would seem to stutter and sometimes even teleport short distances when moving, leading to highlighted issues with peeker's advantage and an overall rough experience. This was further highlighted if the client had poor network conditions. The goal was set to fix the erratic movement of remote characters and make the movement visually smooth.

The original implementation simply interpolated remote characters' positions between two states, one previously received and the latest received positions. Interpolation will be explained in more detail in Chapter 4.3 This type of plain interpolation worked well under near-perfect network conditions and was adequate under decent network conditions, but anything worse than that, the clients would start to have an unpleasant experience. The main issue with this way of handling movement was that it did not consider network variables, such as varying intervals network packets would be received. The remote character would move a constant distance over varying times, meaning a fluctuation in speed. Additionally, dropped packets could even cause the remote character to suddenly teleport short distances.

4.2 Project Scope

The scope started as improving already existing interpolation, implementing an interpolation buffer that would only need changes client-side, and making improvements on an existing replay tool to allow easier debugging of the changes. It was quickly noticed that this would not work on its own and needed supporting fixes to be made on the server's side. Eventually, the scope included reworking the client-side interpolation buffer, adding a running timer that is synchronized between client and server, adding buffering on the server, reworking the way the client sent reliable messages, reworking the way the server handled reliable network packets to work with the buffer, reworking the way fixed update was handled on the server, small fixes on character animations and adding many metrics to help keep track of the game's performance. A detailed explanation of interpolation is found in chapter 4.3.

The buffer in essence normalised the varying time intervals at which network packets were received. Interpolating from the buffer, which contained much more stable data, allowed the remote characters to move at intended speeds, unaffected by network delays. Likewise, the buffer absorbed errors caused by dropped packets, removing the issue with sudden character teleportations. These minor sounding improvements removed uncontrollable randomness, in turn improving the ability for players to acquire targets and removing some peeker's advantage caused by enemies teleporting into view from behind obstacles. Buffering hides some tell-tale signs of bad network conditions, making it feel like network conditions have improved.

Introducing delay by buffering has drawbacks. The added delay accentuated peeker's advantage in some situations. Even with some signs of bad network connection suppressed, the increased delay on top of network delay can cause clients with bad connections to have very noticeable delays in some of their actions.

4.3 Technologies Used with Netcode Improvements

Critical Ops being a published working game included everything needed to implement the changes covered by this thesis. Client-server communication needed no changes and therefore will not be examined in detail in this thesis.

Reliable and Unreliable Communication

As mentioned in chapter 3.4, messages sent over a network are subject to network variables. Some messages in online games are not critical to the function of the game and can be allowed to get dropped, while other messages are vital to the function of the game and should always be received on the other end. [26.] These are called unreliable and reliable communication respectively.

Unreliable messages do not take any precautions to make sure they reach their destination; therefore, the receiving end may never know there was something they were meant to receive [26]. This protocol in games is often used for very frequent messages where a single lost packet does not affect the game much, such as movement. Movement requests are sent out every frame by the client. The server not having information of one in hundreds of movement requests does not usually cause desynchronization between client and server, and what small differences are caused by this can be compensated for as long as they are not allowed to accumulate.

When sending reliable messages, the end stations take precautions and send verifications to make sure the packets are received, intact, and in the correct order. These messages include much more data in their messages and thus increase the amount of bandwidth required. These messages, like any others, may be lost. In these cases, they are sent again until the receiving end has confirmed that the message has been secured, producing larger possible delays. This protocol in games is often used for infrequent messages that have severe consequences if the other end never receives them, such as shooting, especially in a shooter game. When a player shoots, they expect their decision to have an immediate effect. It will lead to a very bad experience if the server never acknowledges the action.

Update Loop

With interactive media, it is important to constantly update the medium with the user’s interactions, this is where the update loop comes in. The interactive media addressed in this thesis will be video games. [27.]

An update loop is the core of any modern video game. Simply put, the update loop is a repeating part of code that checks the player inputs and updates the game state accordingly. Depending on the type of game, the update loop will continue to run even if the player inputs nothing, keeping the game world “alive” and independent of the player’s actions. [27.]

If a new update does not happen, nothing new can be drawn on screen making the game appear frozen, which is why it is also referred to as “framerate” [27].

The rate of the update loop depends on how fast the device currently running the game is, and how much needs to be processed in the current iteration of the loop [27]. This can cause the next iteration to come at uneven intervals. This small deviation in processing intervals is not an issue for most things, but some elements such as physics and keeping several different games in sync need a more reliable interval. This is where a fixed update comes in.

Fixed Update

Fixed timestep or fixed update has been and will be mentioned many times in this thesis, for it is a very important concept in games and especially in online games. Fixed update in essence is an update loop that guarantees a predetermined constant refresh rate for some aspects of a game. [28.] In the case of Critical Ops and many online games, a fixed update is used to update the game state the same number of times between the game and the server, thus avoiding clients and servers running at different rates and desynchronizing over time.

A fixed update can be implemented in many ways. The most common one is with the help of a time accumulator in an update loop [28]. Below in listing 1 is an example of a fixed update implementation inside of an update loop with a time accumulator. Delta time is the difference in time between the previous and current iterations. The variable timeAccumulator grows every update loop by the update's delta time. The FixedUpdate is processed, and timeAccumulator is decreased by the fixed update's delta time. Here is what differentiates this from the regular update, the fixed update is iterated over again and again if there is enough time in the timeAccumulator. Unlike with update where the delta time is the actual interval of iterations, FixedUpdateDeltaTime is a fixed value that FixedUpdate uses for calculations. This way does not guarantee a completely fixed interval for the fixed update, but it does guarantee that a fixed update happens a fixed number of times over a certain period.

```
Update()
{
      timeAccumulator += updateDeltaTime;
      while (timeAccumulator >= fixedUpdateDeltaTime)
      {
            FixedUpdate(fixedUpdateDeltaTime);
            timeAccumulator -= fixedUpdateDeltaTime;
      }
}
```

Listing 1. Simple implementation of a fixed update with a time accumulator

Fixed update is often used for updating the state of the game, such as physics or networking, as the fixed delta time's constant value makes it a more reliable option [28]. Since a fixed update runs independently of the game's other updates and is responsible for updating the state of the game, a single fixed update iteration is often referred to as a “game tick” or “tick”. The regular update loop is responsible for updating visuals, a single update iteration is often referred to as a “frame”. Because the update is responsible for the visual side of things, while fixed update is more “under the hood”, the update usually runs faster than the fixed update. To compensate for the difference between visual updates and simulation state updates, interpolation is used. Interpolation is dissected further below.

Tick Rate

Tick rate is the rate at which the game simulation state is updated, measured in hertz [29]. In other words, the tick rate is the game's fixed update rate, and a single tick is a single fixed update iteration, as mentioned in chapter 4.3. Both offline and online games use ticks to keep a stable simulation of the game isolated from the framerate. In online games, the server ticks are often accompanied by sending game state snapshots to clients [30]. Many games choose to keep the server tick rate and snapshot send rate separately in order to have the game simulation run sharp and have more control over network traffic.

When a player presses an input, the input is processed, and the simulation is updated only on the next tick. If only the tick rate is taken into consideration, a tick rate of 10 would mean that in the worst-case scenario, the delay between input and simulation update is 0.1 seconds. If physics are updated only every 0.1 seconds, an object moving at 10m/s would move 1 meter every tick by appearing at the next position, therefore it may move inside another before it "notices" that a collision should have occurred. When conversing with a server, the delay caused by the tick rate is increased by every processing phase from client to server to remote client, tallying a delay of 0.3 seconds in this scenario. this last issue is explained in more detail in chapter 3.3.

A tick rate of 128 would in turn mean an input delay of ~0.0078 seconds at worst. A physics step for objects moving at 10m/s of ~0.078 meters. And finally, a total of ~0.023 seconds of delay in client-server-remote client sequence.

If a high tick rate is so good, why not have all games run at a tick rate of 256 or something much more? A high tick rate means less time for the computer to process everything, requiring much more processing power, which would in turn cost substantially more money, a scenario that is not possible for most if not all game developers. Finally, a higher tick rate could also mean sending snapshots of the game simulation much more often. Most people do not have enough bandwidth to handle such network loads.

Gamers tend to criticize low tick rates on games as higher tick rates tend to lead to better gaming experiences [30]. Developers have to balance limitations and the object of the game to find a tick rate that suits the project best. Critical Ops ended with a somewhat low tick rate of 30 for an FPS game. The tick rate works for Critical Ops but is not perfect.

Movement Interpolation and Extrapolation

In mathematics, Interpolation refers to estimating a value between known values. [31.]

As explained with fixed update, a game often draws items on the screen in an update loop more often than it updates positions in the fixed update. If nothing is done to account for this, the movement of objects would look very stuttery. Games for this reason often interpolate between object states.

Figure 3 is an example that demonstrates a fixed update interval of 0.1 seconds and an update interval of ~0.03. With a speed of 10m/s, an object moves 1 meter every fixed update iteration by appearing in the next position. An object teleporting forward 1 meter every 0.1 seconds would be a very rough visual. By interpolating between the states every update loop would mean the object snaps forward ~0.3 meters every ~0.3 seconds, giving the appearance of a much more graceful movement.

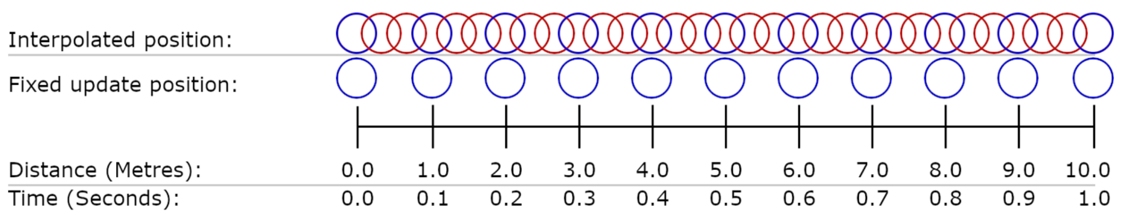


Figure 3 Interpolation is used to smooth out sparse position updates.

Within the context of remote character movement over a network, interpolation is very important. It is used to smooth out the movement between the somewhat infrequent rate of received network packets. This also has the benefit of absorbing some issues brought on by network variables. For instance, state updates from the server are received at varying times due to jitter. If the game would solely rely on these times for updating positions, an object would move a fixed distance over varying times. By interpolating between two states, object movement speed can be normalized as the game would know how much the object should move in the timeframe.

Interpolation only works when the end of the current data sequence is known. If the endpoint is not known, data extrapolation, which is the act of estimating a value beyond known data from an established sequence of data [31], would be needed. Critical Ops extrapolates positions based on previous movement on the server to compensate for lost states server-side.

Interpolation, while useful, has limitations and often does not work well enough on its own. If the network packet containing the next state of an object is either delayed too much or lost, resulting in circumstances where interpolation does not have an endpoint to work with, making interpolation impossible. To make Critical Ops resistant to these kinds of situations, state buffering was introduced to work along with interpolation.

State Buffering

To fortify interpolation against network variables to ensure a better experience for players, buffering needed to be introduced. The buffer is a queue of states on the client, received from the server. The buffer stores states received from the server until the client is ready to use them. By introducing a buffer, the game ensures that interpolation will have a data endpoint to work with every time, barring egregious circumstances.

Critical Ops' original implementation of interpolation only used the previously received state and the latest received state, then interpolated between them using the time between the receiving of the two states. Since network packets may arrive late or not at all, this method was fragile to bad network conditions. A late packet increased the interpolation time and a lost packet forced extrapolation which could miscalculate the actual state of the player, eventually going too much out of sync and forcing a position reset, which meant teleporting the character to the server position.

Storing states before processing them, and then processing not the latest, but a few received packets "in the past" made sure that the time between each state could be normalized, ensuring the character would move a fixed distance over a fixed duration. Having states stored and interpolation times normalized also has the benefit of being able to interpolate over a lost state since interpolation can take the next present state from the buffer as the new endpoint.

Buffering improves character interpolation by a large margin and removes the numerous resets the original implementation had. Resets still occur if the client gets too out of sync with the server, but much less and only with very bad conditions. Buffering is not all positives, the introduced delay increased the issue with peeker's advantage as explained in chapter 3.3, but this is an acceptable price for the many improvements buffering brings.

Jitter Buffer

The Critical Ops game server was made more resistant to network variables with the addition of a jitter buffer. This jitter buffer is a technology used in Voice over Internet Protocol (VoIP)

VoIP is a system that uses the internet to transmit voice and other multimedia content, instead of a regular dedicated line like those made with phones. The media is recorded by the software, packaged into smaller messages, and then sent over the internet to the receiver. [32.] While this technology is mostly used to refer to phone services linked to the internet, it is still used by applications such as Discord or Zoom and is built into some games to facilitate voice communication during a game. [33.]

Why VoIP is important to explain is because transmitting voice over an internet connection suffers from the same issues as transmitting movement over a connection. Movement in a game in a way can be thought of as another form of multimedia content. Both being subject to variables along their journey over the internet to reach their destination, some packets can take a different route or get caught in congestion, causing these packets to take longer to reach their destination and get delayed.

Imagine this, saying “A quick brown fox jumps over the lazy dog” to someone. It is expected that the receiver will hear the same combination of words that make up that sentence in the exact same way. Right? Now imagine the same sentence being divided for several people, and then each person taking their own route to reach the receiver. Some might take the scenic route, some may hit a traffic jam, and some may even get lost and never reach the destination. After this debacle, the words at best are no longer evenly spaced, making the sentence sound weird, and at worst the sentence will be a garbled mess, where the words are no longer in the same sequence and some words are missing. For the same reasons as the above, transmitting movement needs to be done in an orderly fashion. An object moving from A, B and then C cannot be displayed to the receiver as B, A, and then C. It would make no sense to the viewer.

This brings us to the jitter buffer. This system is applied to normalize a signal. When viewing transmitted content, it must be done at a fixed frequency for the content to make sense. Transmitted content is never received at a fixed frequency because of network variables. A jitter buffer buffers a certain amount of data from the signal before passing them on for processing. The processing unit then can retrieve the signal at a steady interval from the jitter buffer. This buffering introduces an artificial latency to the system, where more buffering equals more latency, but in turn is more resistant to network variables. This is a common application in VoIP systems and is also used in video games. [34.]

Critical Ops already had a system in place that made sure that received messages were in the correct order, which made the implementation of this jitter buffer easier. And unlike in transmitting voice, a few lost packets were not an issue, because interpolation could compensate for some missed positions in movement client-side and the server could compensate by extrapolating the missing states.

4.4 Mobile Game Development Limitations

During development, one of the main factors that need to be taken into consideration is the platform(s) the game is being developed for. The different platforms bring different requirements. Console development has a fixed number of platforms that offer known predetermined hardware. PC development has the widest range of hardware variables, and the developer can choose the range the development takes place. Though modern mobile devices are quite powerful, they still fall on the low-end, performance-wise. Mobile gamers may also rely on their mobile data for online gaming, which can be quite poor.

Hardware Limitation

Most modern online shooter games' servers have a tick rate of 60 and some have up to 128. Having a higher tick rate means having faster, more powerful servers. [35.] Critical Ops being a mobile game and in a much smaller scope than its more popular console or PC platform cousins, had to compromise for a much lower tick rate of 30.

Network Limitation

Bandwidth is the most limiting factor when it comes to network protocol, especially so with mobile game development. Critical Ops’ servers originally sent out a network update at 16Hz, which was then increased to 30Hz with the netcode rework discussed in this thesis. A higher rate of network packets meant higher network traffic. While many people are equipped with high-speed internet, the truth of the matter is that most people do not have access to fast and stable internet. This issue is highlighted further with mobile connections.

Limiting the bandwidth used by the servers was crucial to make sure that even players with less-than-ideal connections could enjoy Critical Ops. If the amount of data is not kept in check it could cause network congestion for the players.

An internet connection can be thought of as a pipe, then bandwidth would mean the width of the pipe. The wider the pipe the more data can be pushed through. Congestion is what happens when too much data is being pushed into the pipe, causing a “blockage” which causes high levels of latency, jitter, and packet loss. When streaming video, downloading files or just simply browsing the web, latency, jitter, and packet loss are an inconvenience. However, low latency, minimal jitter, and the least amount of packet loss are paramount when playing online games that have a high tempo, where fractions of a second can make or break a situation. [36.]

In many cases, frequently sending snapshots from the server is vital, in this case, the area where the network load can be lightened is within the snapshots themselves. Being mindful about what information is being sent, what information needs to be sent often, and what can be sent more sparsely can often lead to major savings in bandwidth and is usually enough. In cases where this is not enough, a snapshot can be compressed when sent and decompressed when received. [37.]

5 Implementing Netcode Improvements for Critical Ops

Among the first things to come up during development was how to test this feature. Much of the improvement was on how the game will feel and look after, making code-based testing tricky in some cases. The game could already be set up to work with a local server and test it through that, but this feature was to improve player experience with bad network conditions, and a local network is almost as close to perfect as possible. Setting up a remote server was very time-consuming and therefore could not be used when testing small changes, not to mention that the network conditions still weren't bad enough to push the improvements to their limits. Artificial network conditions needed to be introduced.

To the fortune of the developers, a replay feature had been started for use in debugging. The replay tool recorded packets received from clients, this meant that a replay already simulated an online match but with zero latency and packet loss. Luckily changing the replay tool to include these variables was very simple and could be used with ease to test the current improvements. Later speeding up and slowing down the replay was added to catch smaller visual issues. Further developing the replay tool also had the benefit of it being more useful for others to use for testing, and possibly in the future introducing a replay feature to the players.

Much to our dismay, some small differences were noticed between the replay and live gameplay. Meaning that the replay tool could still be used for testing but could not be relied upon entirely. The clumsy tool was found and solved these issues. It could simulate artificial conditions when playing on a local server, meaning the feature could be easily tested with extreme network conditions. The tool also allowed us to test extreme conditions when playing on a remote server, making it possible to test with more users and see the effects from another point of view.

5.1 First Steps Toward Smooth Movement

Critical Force had an internship for four people, after which the interns were moved to work in different areas of Critical Ops. The author of this thesis who was one of the interns, was moved to work on the netcode and server side of the game. At this point, the development had taken its first steps already with the implementation of an interpolation buffer. The implementation used the original interpolation as its base and added a very simple buffer. This already looked to be a considerable improvement over the original, but thorough testing needed to be done.

At this point, much of the feature’s scale was not realized, so the focus was mostly on testing the buffer and seeing if other small improvements could be made. Fortunately, the replay tool had been developed into a working state and could be made to accommodate this feature’s testing.

Replaying for a Better View

The replay tool’s ability to slow down the time scale was paramount in closely inspecting the movement for flaws. It was already known that harsh network conditions needed to be simulated in order to test the robustness of the buffer, the replay tool was easily modified to include delay, jitter, and packet loss. The first implementation for these used a unity coroutine that sent a network packet from the replay file to the remote characters. This delay between the file and character was then changed depending on the desired simulated network delay. The coroutine’s update rate was increased or decreased with a randomized value within a predetermined interval to simulate jitter. Unreliable packets were then randomly ignored to simulate packet loss.

Fight Against Time

Closely inspecting the character's movement with the replay tool exposed more flaws in the movement. When a character was supposed to move at a completely constant velocity, small acceleration, and deceleration could be seen constantly. This issue was caused by the fact that the interpolation used the received time of the packets for the calculation, and due to jitter, this time was unreliable. Jitter caused the packets to be received at irregular intervals causing varying interpolation times, but the distance travelled between packets remained constant. This meant that the same distance was interpolated over different times. The packet’s send time was then included in the message, for use in the interpolation calculation.

The added timestamp increased the size of the network message by a small amount, but even that small change had to be scrutinized to be sure it would not cause issues with bandwidth.

The client then needed to use its own timer to interpolate between the server’s timestamps. This change would turn out to be a part that needed to be reworked constantly. Two computers running a clock will experience a phenomenon that is referred to as “Time Drift”. The two clocks would not run at the exact same rate and over time would deviate from each other or “drift” apart. [38.] To initially fix this the client would continually reset itself to match the server time. This way of synchronizing was adequate for now, so development would not stall on this matter for too long.

Further Examination

During this time in development, the replay tool proved invaluable not only in testing changes but visualizing the changes to others as well. To hammer the point home, side-by-side comparisons were made using the same replay file, but with the old and new movements enabled. The new movement was still in rough form at this moment in time, but the improvement was still clear and raised interest inside the company.

Changing the Interval

As mentioned above, Critical Ops’ servers sent state updates about remote characters ~16 times each second. This sparse rate of updates needed to be improved. As mentioned in earlier, some games kept their tick rate and snapshot send rate separately, and luckily this was also the case in Critical Ops and this change was simple to implement. For more detailed explanations of the effects of tick rate go to chapter 4.3.

The server ran at a tick rate of ~33Hz, but only sent out snapshots every second tick, translating into a snapshot rate of ~16Hz. This meant that to improve the snapshot rate, the server only needed to send a snapshot every tick.

The snapshot rate was increased to ~33Hz, in turn effectively doubling the bandwidth requirement for the game. The change was deemed acceptable, as the bandwidth requirement for the game was already low. Discussions about further increasing the snapshot rate and the server’s tick rate were also had, but those changes were thought to be too much for the time.

Along with the package interval change, changes to the way the server’s fixed update were processed were made, making it more reliable but somewhat more taxing to the machine. Having these small changes caused performance issues with the server was worrying at first, but it was then quickly discovered that the machines used for the test servers were much less capable than the actual servers. This prompted the creation of an isolated performance server that was used from this point forward for all the non-local tests of this feature.

Adding a Server Buffer

Not much mind was given to the server at first, as these changes were only visual, i.e., only needing changes client-side, therefore changes had only been made on the client end and the movement still did not look the way it was expected. So, attention was shifted to the server and how packets were handled there. The issue turned out to be an amalgamation of a few different methods used by the server.

First, the game was client authoritative. This meant that the server relied on clients to tell it how the game world was changing. Each client would tell the server how that individual changed their game state. More detail on client authoritative architecture is in chapter 3.2. The server could update a character’s position accurately only if the update was received from the client in question on time. If the packet got lost or delayed, said client would not get updated on the server, and could not have an updated position sent out to other clients. Packets sent by the client will be referred to as client “requests” because even though in a client-authoritative architecture the client tells the server what is happening, the server does not need to accept everything.

Second, to solve this problem originally, Critical Ops uses extrapolation to predict where a character would move based on their previous trajectory. This prediction however could not always be completely accurate, especially if too much extrapolation takes place, since in a game where the trajectory of a character could be changed drastically in each frame, large amounts of extrapolation is very inaccurate. Extrapolating a character’s position incorrectly would lead to a snap in position when the server receives a packet with the character’s actual position.

Third, network messages are queued and sent out in small batches. Meaning the server could receive several position updates from a client at once and have moments with no updates received. With no queue in place to handle these batched position updates, meant that the latest position would override all others, and the characters would idle until new updates were received. For example, the server receives position updates A, B, and C for a client on tick 3. The server would then apply all three position updates in the same tick on tick 4, effectively moving the character three times the distance in one tick. Then, while receiving nothing on ticks 4, 5, and 6 the character would idle until once again receiving ticks D, E, and F on tick 7.

Jitter being the largest issue affecting movement on the server, A jitter buffer was implemented for much the same reasons it is used in VOIP, that is, to give the server time to clear and normalize the received data. More on this subject can be read in the chapter 4.3.

Open for Testing

With the numerous changes to both the client and server, it was time to let the feature out to a wider audience to test, after all, there is only so much two people can see in their own work in a short amount of time. And many new issues were revealed.

An internal company-wide test session was held. The goal of this session was to see if people could feel an improvement and to find out what else still needed improvement.

It was quickly documented that the server failed to validate client actions and rolled them back often. While testers complimented on the quality, it was clear a lot of work still needed to be done. And to the eyes of this feature's developers, the movement did not quite look to be all the way there yet. All in all, the session was a success. Improvements were clear and liked, and new goals were set.

So far, the changes made have been numerous and large. Going forward the changes will seem smaller, and the number of changes will be lower. This was a consequence of jumping into the work on this feature while being unfamiliar with the code base and the fact that this first iteration took much more time than others. Most of the improvements mentioned above required reworking down the line when new discoveries were made.

5.2 Accommodability Through an Adaptive Jitter Buffer

Fixing Things

After the first large test session, some time was taken to investigate the cause of the issues found there. Time was also taken to better understand what changes might be needed to further improve the movement.

The first thing to do was to fix the broken validation. It took some time to find out that when validating shoot requests, timestamps in the packets were used to check if they were sent at a realistic time, meaning the timestamp could not be newer than the latest processed movement request nor could it be too old compared to it. The problem originated in the jitter buffer. Movement requests were now being buffered and processed at a later point, while other requests such as shooting were processed immediately when the server received them, sometimes causing a shoot request to be processed earlier than its corresponding move request.

To fix the validation issue with timestamps, two systems were introduced. One was applying all gameplay-related reliable requests on the server to the jitter buffer. The second was to create a queue on the client for sent messages. Before the client could send out a movement request at the end of a tick, a shoot request may have been sent during a frame in between two ticks. The fix was to simply buffer all requests and send them out at the end of every tick. This unfortunately required other systems to be implemented.

Pooling Objects

An object pool was added to support the queue. The purpose of this was to avoid constant unnecessary memory allocation which would cause performance issues with C# garbage collection. All requests also had to be reworked to be of a matching type for the queue to work.

Changing the Jitter Buffer

At this point, it was decided that the jitter buffer needed to accommodate a wider range of network qualities, so it was changed from a fixed jitter buffer to an adaptive jitter buffer. This meant that instead of having a fixed amount buffered, i.e., working well on a certain network, working suboptimally on another, and not working at all on some, the jitter buffer would now adapt to each client’s network by calibrating values based on certain variables. This change made the jitter buffer much more robust against jitter and more suitable for a wide range of users.

Fight Against Time

As was already suspected, the time synchronization issues returned and needed improvement. The existing system would cause sudden small jumps in the client’s clock, which would manifest to the players through a sudden acceleration of remote characters' movement. This was not always visible but did not give the impression of a finished polished game. Exponential smoothing algorithm was implemented to smooth out the time drift.

$$Offset = \mathrm{smoothFactor} * \mathrm{serverTime} + \left((1 - \mathrm{smoothFactor}) * \mathrm{offset}\right)$$

The client would return to having its own running clock, and an offset would be added to that time based on the server’s time. The smoothFactor in this algorithm is an arbitrary value used to limit how much the offset changes each iteration. Using this algorithm, the time drift would be compensated for every frame, instead of every tick, removing many sudden jumps in the clock synchronization. This algorithm would also return higher values when the difference between server and client times was larger, or smaller values when the difference was smaller, meaning the algorithm would correct more if the drift was larger.

Further Testing

With these changes, a second test session was held, with the same goal as the first test session in mind.

Immediately the developers noticed very small stuttering in the movement. Something that was not completely visible to anybody just looking at the game but did not hold up for close inspection. Shooting no longer caused constant failed validation, but validation still did not work on weapons with multiple projectiles. An issue with animations was also spotted, If a client would disconnect while in an animation, the animation would loop constantly until they reconnected.

5.3 Performance Profiling

Number of Fixes

Multi-projectile weapon validation issue was quickly fixed. The number of projectiles was not properly conveyed between the client and the server. The server did not update a character in any way if no new messages were received, leading to the issue with the disconnection animations. Several other small fixes and tweaks were made along with these fixes.

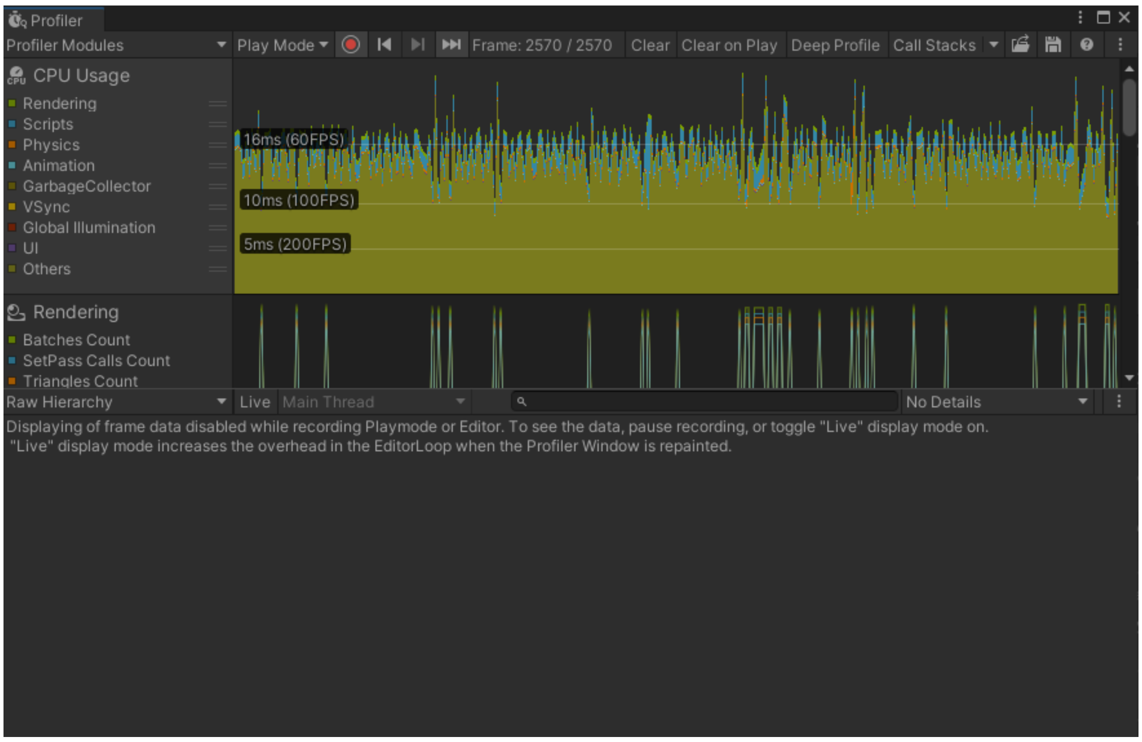
Time Squandered

While working on the validation fix, an issue with grenades was spotted. In some rare cases when a grenade was thrown, an “empty” grenade would remain in the character’s hands that did not function. This issue seemed like a major one, as it directly and clearly interfered with gameplay. The detriment in fixing this was that there were no clear steps to reproduce the issue, it seemed like it would happen randomly. Every time it appeared like this was reproduced, performing the same steps would not recreate it. Much time was spent on trying to fix this, until it was noticed that the issue was extremely rare and existed in other branches as well, meaning that the netcode improvements were not the cause of this. Development moved on from this.

Unity Game Profiling

Focus was then shifted to profiling client performance. Critical Ops, being a mobile game, had to cater to low and high-end devices trying to run the game. Not taking steps to ensure low-end devices can run the game would mean that a large part of the audience would be cut off.

The Unity Profiler tool was used extensively during this. While the effects on the performance could be seen with the profiler tool, nothing significant enough was documented to warrant further action. The hopes were also that fixing a performance issue would also fix the small stuttering in the movement. To make matters worse, the Unity Profiler tool would display the performance load caused by the editor itself as seen in figure 4, making profiling slightly more difficult.



Displaying of frame data disabled while recording Playmode or Editor. To see the data, pause recording, or toggle "Live" display mode on.
"Live" display mode increases the overhead in the EditorLoop when the Profiler Window is repainted.

Figure 4 View from Unity Profiler. Unity editor causes spikes in the profiler.

More Testing

With these changes in hand, another test session was held. This time the stuttering in the movement was even more subtle but still present. An older issue was documented to have been made more prominent with the netcode improvements, pushing others out of the bounds of the map. The out-of-bounds issue had always existed in Critical Ops but could only be performed under certain circumstances. With the netcode improvements, this could happen much more often.

5.4 Performance Metrics

Out of Scope Issue

The out-of-bonds issue was reproduced with ease and considered to be yet again a large issue. Much time was spent investigating what could be the cause of it, and it was eventually pinpointed to be the delayed processing of remote character states due to buffering. Because of the increased delay, remote characters may not register a collision on their end while pushing local characters around.

A new initiative was then created to try and fix the out-of-bounds issue. Development of this issue went as far as putting together a team and some small testing to see what could be introduced to fix it. Frustrated with how far this initiative would diverge from the original netcode improvements, other teams were contacted to figure out exactly how big the issue was. It was then discovered just how common the issue was and could be fixed later to not take focus away from the netcode improvements.

Tracking Performance

Small tweaks and fixes were made on the server to increase the performance of the jitter buffer. To find out exactly how the jitter buffer performed, metrics were introduced for the server to be displayed on the cloud monitoring service Datadog.

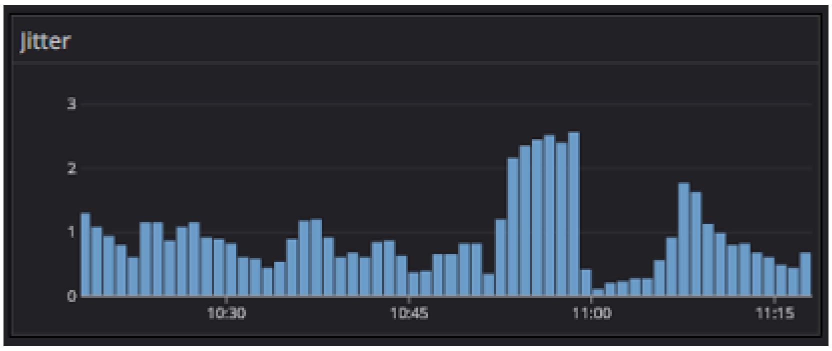


Figure 5 A single metric widget from Datadog. Keeping track of different client's performances was easier with Datadog.

The initial implementation for the metrics was simple. To deal with concurrency issues, a service with thread-safe counters was injected into the jitter buffer, which in turn would increment or decrement the necessary values when required. These values would then be sent to Datadog for processing. Values such as average jitter as seen above in figure 5 could then be displayed in an easily readable way. This enabled a way to keep track of the performance of the server and clients.

Another Test

When hosting a test session at this point, no large issues were spotted, and many were satisfied with the improved feel of the game. The previously present small stutter in the movement was however still present and could not be ignored.

5.5 Synchronizing Time

Correcting Time Drift

The small stuttering was suspected to be caused by the way time synchronization was handled by the client. During this time the time synchronization went through several iterations and much ideation. Work was done to minimize the time drift, but without a properly synchronized clock between client and server, this issue would be ever-present.

Eventually, it was decided that the time synchronization needed to detect when it had drifted apart enough, and then proceed to correct for the drift over a predetermined duration. This was to minimize the number of corrections performed, and when corrections did need to occur, they would be as unnoticeable as possible.

The current times between server and client were used to detect when they had drifted apart enough from each other to warrant correction. Incoming raw values were too unstable for detecting time drift due to jitter, a packet could have been 10 milliseconds early on one tick, and 10 milliseconds late the very next. The exponential smoothing algorithm was used to sort out deviant values caused by jitter. When enough drift was detected for correction needing to happen, the required amount of correction would be calculated and applied over the course of a set duration by dividing the amount of correction needed evenly across every frame for the duration of the correction.

Optimizing Metrics Collection

The performance of metrics collection then came into focus. The implementation at the time was not optimal and required a complete rework. Sending a lot of raw data was good for the integrity of the statistics especially when it was not completely certain what data and in what form was it useful. To lessen the burden on the server's bandwidth less data needed to be sent. Much thought was given to what data was needed and how it should be processed before sending to maintain the validity of the dataset.

Each game would now collect information on each client. The data collected from each game would then be aggregated. From this large collection of raw data, percentile calculation would be used to eliminate outliers. Jitter was used as the metric to determine a user’s network quality, and based on that, high, medium, and low-performing clients’ data would be sent to Datadog.

Testing and Still Stuttering

Another wider test session was held. No new issues were found by the testers. The small stuttering in a character’s movement unfortunately remained.

5.6 Getting to the Root of the Issue

Stutter Fixed, Once and for All

Ideas on what could cause the stuttering were running low. Thus far, many suspects were eliminated, and nothing did more than slightly improve it at best. The character’s object hierarchy was then examined more closely. During gameplay, a visible shake could be seen in some objects’ positional coordinates. Then by what might be just pure luck, it was spotted that root motion on animations was enabled and that the code was applying movement based on animations as well. The use of this root motion turned out to be a remnant of previous versions of the game that was long since removed.

Animation root motion is motion in an animation that is applied to the root transform of the object, which is then translated into movement for the whole object [39]. This root motion was fighting against movement being applied to the object through code, which caused the stuttering in the movement.

With limited experience in animations, and much more experience being easily available, other people inside Critical Force were consulted. With the help of other teams, this matter was a trivial fix. And just like that, the movement was finally rid of the ever-present stutter that plagued development for so long by removing all root motion from the animation files.

Test Session

Another larger test session was held not long after. With all movement-related issues finally fixed, much positive feedback was given. An issue was still found with characters' spawning in certain game modes. The characters would not update their vertical position during a freeze time at the beginning of certain game modes and would appear to either float or be partially underground. Characters were also sometimes spotted spinning rapidly for no apparent reason.

5.7 Final Improvements

Fixing Input Handling

The issue with character spawning was figured out to be caused by improper handling of inputs on the server during freeze time, which took place at the beginning of certain game modes. The game being client authoritative meant that the server had to rely on the client for physics calculation. During freeze time the vertical position updates were mistakenly disabled as well, when they should have been enabled to allow characters to fall and move out from inside objects.

The character spinning was discovered to be caused using a cache value on the server when shooting, instead of the proper queued value. This cached value is used when receiving a new request from a client to avoid constantly needing to allocate memory. Applying the cache value instead of the queued value meant that the latest request received by the server would be applied no matter whom it was meant for, instead of the actual request the character should perform at that moment, applying the wrong values to the wrong character at the wrong time.

Analysing Code

After fixing the problems that came up during the last test session, no more clear goals were available, so improving the existing code was the logical next step. The tool NDepend was used to analyse the code base and see if there were issues or tech debt introduced by the rework. The tool was very useful, and many improvements were made to reduce tech debt.

Close to the Finish Line

A final test session was held with confidence high. All known issues had been addressed, and in small-scale testing, the movement looked very good. When the time of the test session came along, the feelings of confidence were validated when no issues were found, and everything worked. It was time to introduce the rework to the players in public beta.

Summary of Fixes

- Client-side
  - Interpolation was updated to use server time for a more stable interval.
  - Buffering was added to support interpolation.
  - Replay tool was updated to support testing.
  - Game state snapshot rate increased.
  - Reworked how the client sent messages by batching and sending all reliable messages at the end of each tick.
  - Client clock was synchronised with the server clock.
  - An object pool was added for the requests.
  - Performance metrics were added.
  - Animation root motion was removed from animation files.

- Server-side
  - Added a jitter buffer.
  - Reworked reliable request handling.
  - Added object pooling for reliable requests.
  - Reworked fixed update.
  - Game state snapshot rate increased.
  - Added server time to the messages sent to enable time synchronisation with client.
  - Added performance metrics.

6 Recurring Challenges Faced During Development

6.1 Growing Scope

The scope of the feature was underestimated and kept growing constantly as new systems linked to each other in need of change were discovered. Still, all of the changes made were necessary to have the movement in its current improved state. Working with a large and fairly old code base that neither developer was completely familiar with led to finding more work as the feature progressed. Even with more experience, this would have been an unavoidable issue but could have been mitigated to an extent with a deeper knowledge of how different systems were connected. More planning and research at the start of the feature would have improved the scope estimate somewhat, but seeing how everything progressed, this was an unavoidable outcome and could not have been improved by much.

6.2 Time Consuming Testing

Constant testing was done rigorously throughout development. When implementing new code, updating, adjusting, and otherwise iterating on existing parts, testing was done after every change. Frequent tests were made to make sure everything worked as intended, and if made adjustments that were made improved or impaired the game. Since changes were made in both client and server, both needed to be tested often.

Testing the server was much simpler where the use of unit tests was capitalized on, making most testing for the server a faster process. Unit tests unfortunately were not available for the Unity client version of the game, where running the game locally was the fastest way to test.

Seeing as how the feature improved upon the communication between server and client, a local environment running the server had to be set up when the effects of a small change on the client wanted to be tested. This meant starting a backend environment for the server to communicate with, then game data had to be uploaded to the backend. After that, the local server could be started. Only after that could the Unity client connect to the local server. This would not take too much time, but doing all this many times accumulated a considerable time sink. And all of this was not mentioning the time it took for Unity to compile and run the game.

6.3 Being on Time

Having both client and server in sync was always a requirement, and Critical Ops already managed that. Using the server’s time in the client’s character interpolation meant that the synchronization now needed to be very precise. The moment this necessary change was introduced, it caused constant issues and consequently reworks. Keeping the times synchronized was vital to the function of the movement. If the clocks were allowed to drift freely, remote characters would move slower or faster than intended depending on the direction of the drift. This would continue until eventually, the time would go out of sync enough to break the interpolation, an unacceptable outcome.

While the current implementation with the time drift corrector is a fully functional one, it is a fundamentally flawed one, made to compensate for the lack of an accurate synchronized clock. Even during the writing of this, discussions were had about the changes that could and should be introduced to the time synchronization. The fact that the current time drift corrector would wait until it goes out of sync enough to even start correcting is flawed. And to add to that, the current implementation can only detect and correct a fixed amount of desync at a time, meaning that large drifts must be corrected multiple times in small increments.

7 Results and Future for Current Netcode Improvements

Much support was given to the development of this feature from the company. Not having time pressure and being able to expand the confines of the work as it progressed, made for a very pleasant development. The final product was very well received by people in the company and beta testers alike.

There were times when what needed to be done next was unclear and seeing how the feature kept growing with no end in sight at times felt burdensome. Looking back at how the development went, not much could have been done about that. Some more planning and familiarising with the code base may have alleviated this to some extent, though both developers stated that they often preferred to work more iteratively, rather than try to plan everything out too carefully. Planning everything out would have been a difficult and time-consuming undertaking, and these kinds of plants have a habit of not working out as intended anyway.

After the beta ended, the feature was put up for peer code review. The feature being nearly 6 months of work at that point, made the code review a long and arduous process. To expedite the process, the review was done as a group, with the developers explaining more complicated parts of the code as the review advanced. The code review proved, as always, extremely useful albeit tiring. Many points of improvement were found during the review.

The way the code review was handled, was all in all a very positive experience. Others could gain a deeper understanding of exactly what changes were made and why, while the developers who worked on the feature could refresh their memory on the changes and look at their code through a more critical lens.

7.1 Working with a Pair

Soon after the internship period ended, the work started on this feature. It was also at this point, Critical Force decided to change how teams worked together, with the intention of having smaller teams or pairs that would work closely together.

The team created to work on the netcode improvement feature consisted of Jimmy Meder (Junior programmer) and Nathan Biagini (Network Developer).

With little experience in working for a game company, and especially in network programming, starting was very overwhelming. The number of new systems and the size and age of the code base created a difficult environment for someone with little experience to get a foothold. Working as a pair eased this process.

Pair programming was utilized heavily during the development of this feature. There were times when work was done independently, but every time pair programming was exercised, both developers agreed that it made brainstorming ideas more efficient, fewer mistakes were made and more work got done in the same amount of time.

The experience only had positives and will likely be the way development is done going forward.

7.2 Collected Metrics

When the beta started for this feature, having metrics collection was important. These metrics were displayed in Datadog, which enabled them to be understood just at a glance. These collected metrics, some of which can be viewed below in figure 6, made it possible to keep track of the performance of players and the jitter buffer.

The number of different metrics could still make it unclear what to look at and when as seen in figure 6. While all these widgets are important in calibrating the jitter buffer, pressing issues can be seen with the help of just a few of them. Jitter was used to keep track of how stable the players' connections were, with the best connections having values as low as 0 milliseconds and the worst connections going as high as 100 milliseconds. Input Processing delay was important to keep track of how these changes affected peeker's advantage. Based on the collected metrics, the input processing delay was needlessly high, and the jitter buffer would get calibrated to lower it.

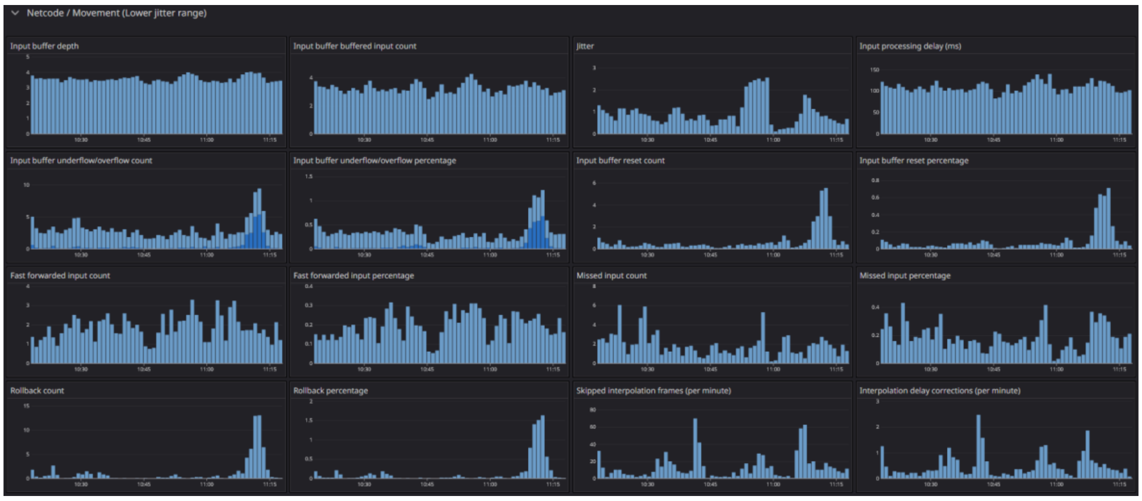

Netcode / Movement (Lower jitter range)
Input buffer depth
Input buffer buffered input count
Jitter
Input processing delay (ms)
Input buffer underflow/overflow count
Input buffer underflow/overflow percentage
Input buffer reset count
Input buffer reset percentage
Fast forwarded input count
Fast forwarded input percentage
Missed input count
Missed input percentage
Rollback count
Rollback percentage
Skipped interpolation frames (per minute)
Interpolation delay corrections (per minute)

Figure 6 Datadog view of collected metrics. Observing for spikes could reveal issues.

7.3 Testing and Feedback

Private-Test-Program

Once development had reached a stable point, it was decided that people other than company personnel needed to test the changes, to get a fresh point of view.

Critical Force hosted a private test session for a small group of volunteer players from different regions. During the test, most players connected with less-than-ideal network conditions, which was perfect for the purposes of the test.

Feedback received from the volunteers came back as very positive. Some commented on how the game felt like their network conditions had improved, compared to their previous experience. When in reality they were connecting to a server much further away.

Public Beta

The feature was in beta from March 1st. To March 27th. 2023. There were up to 64 players on a single server during the beta. A single server had as many as 15 games running simultaneously. Dropping in random games and asking the players how the game felt only garnered positive comments. No reports about Peeker's advantage were made during the beta. Two causes for crashing were discovered during the beta.

Internal Feedback

Throughout development, feedback from peers had been very positive and encouraging. During wider test sessions held at the end of each development cycle, people were very pleased with how the changes affected the game.

7.4 Going Forward

During the code review, many code style issues were found and quickly fixed. In addition to the small fixes, a concurrency issue was discovered in the way metrics collection was implemented and an issue that may enable clients to crash the server. Optimizations could also be done in the jitter buffer's jitter calibration.

As no surprise, the time drift correction between client and server, while functional, was determined to not be optimal, and would require some reworking. A full synchronized clock was ruled out of scope to implement with this feature as it would require a considerable amount of work.

After fixing the existing issues, a synchronized clock could be implemented to improve the interpolation. The synchronized clock will most likely prove useful in the future when implementing other features that require close synchronization between client and server.

The collision issue that already existed but was made more prominent with the changes needs to be addressed in the future as well.

Values affecting the interpolation need to be made configurable and will require constant monitoring and adjusting to suit the players' needs. Other optimizations and improvements are likely to be found in the future, making this a feature that requires work for a long time.

Critical Force is also looking into creating other games based on the Critical Ops' code base. This would mean that every possible future game that uses the base, would benefit greatly from the work done here.

8 Conclusion

This thesis details on the work done to improve netcode in Critical Ops, while being employed as a Junior Programmer for Critical Force. These netcode improvements updated the way Critical Ops handled gameplay related messages between client and server.

Critical Ops' movement for remote characters was improved by introducing buffering on both the client and server. Buffering received messages allowed the processing interval to be normalized, along with improving interpolation calculation.

The changes were made to address the stuttery and unpolished look of Critical Ops' remote character movement. The stuttery movement would also cause issues with aiming and with peeker’s advantage. The changes eliminated the sudden skips in movement, making the game feel more stable.

While the added changes removed some unfairness caused by stuttery movement, especially by removing the occurrence of having characters suddenly appear into view from behind corners, the delay between what the player perceived and what the server processed increased. This increased delay will, in turn, increase Peeker’s advantage, but in a different way. Excluding peeker’s advantage, the smoother movement brought by the new implementation improves the feel of the game across the board.

Some performance issues still exist in the implementation. Even though these are not severe, a better-performing game reaches a wider audience, especially if it’s a mobile game.

References

1 Piirainen, Veli-Pekka. Chairman of the Board, Critical Force Oy, Kajaani. Remote interview 28.4.2023.

2 About Critical Force. Online material. Critical Force Oy. <https://criticalforce.fi/about-us/>. Read 28.4.2023.

3 Heikkinen, Pekka. Producer, Critical Force Oy, Kajaani. Remote interview 28.4.2023.

4 Piirainen, Veli-Pekka. 2013. Critical Missions: SWAT – From a Hobby to a Viral Success. Online material. <https://www.youtube.com/watch?v=mLCEyV7Op_M>. Viewed 28.4.2023.

5 Piirainen, Veli-Pekka. 2017. The Mini-Documentary of CF to Critical Ops. Online material. <https://www.youtube.com/watch?v=gjwCX0rCSMY>. Viewed 28.4.2023.

6 Klar, Jonas. Senior Game Designer, Critical Force Oy, Helsinki. Remote Interview 28.4.2023.

7 Gershgorn, Dave. 2018. The game that kicked off a video game revolution turns 25 today. Online material. G/O Media. <https://qz.com/1490069/doom-the-game-that-kicked-off-a-video-game-revolution-turns-25-today>. 10.12.2018. Read 11.3.2023.

8 Keizer, Gregg. 1994. Doom. Electronic Entertainment 4.4.1994. p. 94.

9 The History of Online Shooters. 2010. Online material. IGN. <https://www.ign.com/articles/2010/01/07/the-history-of-online-shooters>. Updated 10.5.2012. Read 11.3.2023.

10 Larch, Florian. 2023. eSports History: How it all began. Online material. ISPO Sports Business Network. <https://www.ispo.com/en/sports-business/esports-history-how-it-all-began>. 8.2.2023. Read 11.3.2023.

11 Carmack, John. John Carmack .plan Archive. <https://github.com/ESWAT/john-carmack-plan-archive/blob/master/by_day/johnc_plan_19960802.txt> Read 11.3.2023.

12 HALF-LIFE. Online material. Valve Corporation. <https://www.half-life.com/en/halflife>. Read 11.3.2023.

13 Tekin, Barış. 2021. HALF-LIFE: The Game That Changed the Game. Online material. G-Loot Global Esports AB. <https://stryda.gg/news/half-life-the-game-that-changed-the-game>. 1.12.2021. Read 11.3.2023.

14 Cliffe, Jess. 2000. CS V1.0 Released!. Online material. <https://web.archive.org/web/20001201214200/http://counter-strike.net/>. 9.11.2000. Read 15.4.2023.

15 Henningson, Joakim. 2020. The history of Counter-Strike. Online material. Red Bull GmbH. <https://www.redbull.com/se-en/history-of-counterstrike>. 8.6.2020. Read 15.4.2023

16 Roxl, Rhett. 2021. What is Peer-to-Peer Gaming, and How Does it Work?. Online material. VGKAMI, LLC. <https://vgkami.com/what-is-peer-to-peer-gaming-and-how-does-it-work/>. 10.12.2021. Read 14.3.2023.

17 Pandey, Harsh. 2022. Peer-to-peer vs client-server architecture for multiplayer games. Online material. Hathora. <https://blog.hathora.dev/peer-to-peer-vs-client-server-architecture/>. 11.5.2022. Read 14.3.2023.

18 Herron, Mike. 2018. Choosing the right client/server relationship. Online material. ChilliConnect. <https://www.chilliconnect.com/best-practice-01-choosing-the-right-client-server-relationship/#:~:text=Fully%20client%20authoritative%20games%20perform,and%20trust%20the%20players%20device>. 17.9.2018. Read 14.3.2023.

19 Gambetta, Gabriel. Fast-Paced Multiplayer (Part I): Client-Server Game Architecture. Online material. Gabriel Gambetta. <https://www.gabrielgambetta.com/client-server-game-architecture.html>. Read 14.3.2023

20 deWet, Matt & Straily, David. Peeking into valorant's netcode. Online material. Riot Games, inc. <https://technology.riotgames.com/news/peeking-valorants-netcode> 28.7.2020. Read 26.9.2022.

21 Fakhoury, Manal & Schwuttke, Ursula. Reaction Time. Online material. Institute for Human and Machine Cognition <https://www.ihmc.us/wp-content/uploads/2021/03/2021-03-Reaction-Time-2.pdf>. Read 25.4.2023.

22 Behzad, Ahmad. What are the different kinds of computing network delays?. Online material. Educative, inc. <https://www.educative.io/answers/what-are-the-different-kinds-of-computing-network-delays>. Read 3.4.2023.

23 What is a denial of service attack (DOS) ?. Online material. Palo Alto Networks. <https://www.paloaltonetworks.com/cyberpedia/what-is-a-denial-of-service-attack-dos>. Read 3.4.2023.

24 IR Team. Network Jitter – Common Causes and Best Solutions. Online material. Integrated Research Ltd. <https://www.ir.com/guides/what-is-network-jitter>. Read 3.4.2023.

25 Miner, Matthew. What is Packet Loss?. Online material. Matthew Miner. <https://packetlosstest.com/packet-loss>. Read 3.4.2023.

26 Reliable vs. Unreliable Communication. Online material. InetDaemon Enterprises. <https://www.inetdaemon.com/tutorials/basic_concepts/communication/reliable_vs_unreliable.shtml>. Updated 19.5.2018. Read 3.4.2023.

27 Nystrom, Robert. Game Loop. Online Material. Robert Nystrom. <https://gameprogrammingpatterns.com/game-loop.html>. Read 15.3.2023.

28 Fiedler, Glenn. 2004. Fix Your Timestep!. Online material. Glenn Fiedler. <https://gafferongames.com/post/fix_your_timestep/>. 10.6.2004. Read 15.3.2023.

29 Tsiaoussidis, Alex. 2023. How does tick rate work in Counter-Strike 2?. Online material. Dot Esports. <https://dotesports.com/counter-strike/news/how-does-tick-rate-work-in-counter-strike-2>. 22.3.2023. Read 5.4.2023.

30 Lee, W.-K., & Chang, R. K. C. (2015). Evaluation of lag-related configurations in first-person shooter games. 2015 International Workshop on Network and Systems Support for Games (NetGames). doi:10.1109/netgames.2015.7382997. Read 12.4.2023.

31 Awati, Rahul. Extrapolation and interpolation. Online material. TechTarget. <https://www.techtarget.com/whatis/definition/extrapolation-and-interpolation>. Updated 5.2022. Read 20.4.2023.

32 Voice Over Internet Protocol. Online material. Federal Communications Commission. <https://www.fcc.gov/general/voice-over-internet-protocol-voip>. Read 20.4.2023.

33 Jenkins, David. 2007. Pokemon Diamond/Pearl Gets VoIP Headset. Online material. Informa PLC. <https://www.gamedeveloper.com/console/-i-pokemon-diamond-i-i-pearl-i-gets-voip-headset>. 19.3.2007. Read 20.4.2023.

34 Biagini, Nathan. Network Developer, Critical Force Oy, Metz. Remote interview 2.5.2023.

35 Randall, Brent. 2020. Valorant’s 128-Tick Servers. Riot Games, inc. <https://technology.riotgames.com/news/valorants-128-tick-servers>. 31.8.2020. Read 15.4.2023.

36 IR team. A Guide To Network Congestion: What Is It, Causes, and How To Fix It. Integrated Research Ltd. <https://www.ir.com/guides/network-congestion>. Read 20.4.2023.

37 Fiedler, Glenn. 2015. Snapshot Compression. Glenn Fiedler. <https://www.gafferongames.com/post/snapshot_compression/>. 4.1.2015. Read 21.9.2022

38 Time Drift (NTP). Online material. Blue Matador, Inc. <https://www.bluematador.com/docs/troubleshooting/time-drift-ntp>. Read 20.4.2023.

39 Root Motion – how it works. Online material. Unity Software Inc. <https://docs.unity3d.com/Manual/RootMotion.html>. Read 16.1.2023