Universiteit Leiden
The Netherlands

Bachelor Computer Science

ButtonCheck: a Framework to Benchmark the Performance of Netcodes for Fighting Games

Michiel van der Bijl

Supervisor:
Kristian Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

13/08/2026

Abstract

Online multiplayer games have become the dominant force in the gaming space, and as such ensuring the quality of their network communications is of great importance. Fighting games are particularly reliant on network communications to function, as they revolve around having multiple players compete against each other in high precision matches in realtime. Because of this, they need to be strict on their accuracy as to not give any player an unfair advantage over the other. To this end, there are many implementations of game network communications to choose from that seek to remedy this issue, but it can be unclear which one is best suited for the task at hand. This thesis presents ButtonCheck, a benchmark to evaluate netcode libraries on their accuracy. We also present Benchy-Fighters, a generic 2D fighting game with exchangeable netcode implementations. Three netcode libraries are evaluated: Yojimbo, GameNetworkingSockets and RakNet. Our findings reveal that there are significant differences in Yojimbo's accuracy and robustness when compared to the other netcodes tested, Additionally, significant differences were found in CPU time between all netcodes tested. No convincing results were found to indicate a relation between accuracy and CPU time, and Yojimbo is shown to be significantly less robust to certain network conditions. These results show that there are notable performance differences between netcodes, encouraging further research into the reason for these differences and their impact on performance.

Contents

1 Introduction

Video gaming is a massively widespread hobby, and for some players even forms their livelihoods. In this space online multiplayer games have become the dominant force. Steam[1], the biggest PC game marketplace, reports that of the top 50 all-time peak of concurrent players as of August 3rd 2026, 37 games offer a multiplayer mode and 21 of those can only be played with multiple people. The game with the most concurrent players is PUBG: BATTLEGROUNDS with a record of 3.3 million players. Behind that in second place is an exclusively singleplayer game, Black Myth: Wukong. With a peak concurrent player count of 2.4 million, it sits over 800.000 players lower then its multiplayer competitor [1].

When compared to most other game genres, fighting games in particular have always been multiplayer games in nature. With player-versus-player competition at its core, fighting games have been made for social play since their infancy [2]. This aspect has only become more prominent with time, as a myriad of tournaments both in person and online have spread all over the globe [3]. Historically, fighting games were often played in arcades, where onlookers would gather as players competed. As consoles became more prominent, fighting games would move with them into households, where friends could compete with each other [2]. Gradually, as consoles gained networking capabilities with time, fighting games would adapt through online multiplayer functionality.

This move to an online multiplayer environment presents many challenges, as the limitations imposed by online play directly impact the technical precision with which actions can be performed, a skill that is central to fighting games [4]. Lag can make it difficult for a player to properly time their actions, and network instability may hide an opponent's actions, in turn making it harder to respond to them. Many networking strategies have been devised and developed to remedy these issues, such as delay-based netcode and rollback netcode [4].

Over the years, a vast variety of implementations of these strategies has been created and published for public use [5]. These netcodes have been written for use in various programming languages from C to Rust, and in some cases even game engines like Unity. Due to the vast quantity of options and the lack of broadly applicable tools to benchmark them, it is challenging to determine which implementation is best suited for any given project.

The aim of this thesis is to create a framework to benchmark various netcode implementations to evaluate their accuracy. To this end, we pose the following research question and subquestions:

> ***Research Question.*** *How do different open-source netcodes perform in terms of accuracy when benchmarked against various network conditions in Fighting games?*
>
> *Sub-question 1. What variables can influence the accuracy of a netcode?*
>
> *Sub-question 2. How can a framework to benchmark netcodes on their accuracy be designed?*

[1] https://store.steampowered.com

This thesis proposes a benchmark which evaluates netcode libraries on their accuracy. Included is Benchy-Fighters, a simple 2D fighting game with exchangable netcode implementations. Subsequently, three netcodes are evaluated: Yojimbo, GameNetworkingSockets and RakNet. We find that there are large differences in Yojimbo's accuracy and robustness in comparison to GameNetworkingSockets and RakNet, and there are notable differences in CPU time between all three netcodes. No convincing results were found to indicate a relation between accuracy and CPU time, and Yojimbo is shown to be less robust then the other netcodes under the tested network conditions.

The remainder of this thesis is organized as follows: Section 2 describes and defines various terms that will be used throughout this work. Section 3 details related works. Section 4 presents the benchmark and the accompanying game, explaining design decisions and the overall implementation of both. Section 5 shows how data was gathered, how and why the chosen netcodes were selected, the experiments that were performed and their results. Lastly, Section 6 answers the research questions posed here, discusses limitations and considers future works. Additionally, Appendix A contains the specifications of the testing setup and the device that was used, Appendix B contains the raw data gathered in the experiments and Appendix C provides information on how to install and compile the programs used.

2 Background

In this section, firstly we will explain the relevant terminology used to describe a fighting game engine. Then, we review terminology around netcodes and introduce new terminology and definitions that will be used throughout this thesis.

2.1 Fighting Game Engines

At its core, game engines have a simple task: take the given inputs and calculate the results of those inputs. There are three relevant parts to this task: yhe inputs given, the state the game is in before those inputs were given (referred to as a gamestate) and lastly the rules that translate the given inputs into changes to the state of the game.

We define a gamestate as an instance of all the data describing a game world while the game is being played. In the case of a fighting game, this describes information such as what location a player is in, what action a player is performing and how much health a player has. A gamestate does not describe information such as the FPS a game is running with or what language is used to display text. An easy way to understand if information is described by a gamestate is if the information could change through a player's actions in the middle of playing a match.

The actions a player can take differ between fighting games, but generally a player is able to walk left or right, jump, block by moving away from the opponent or attack by performing a specific set of inputs. Blocking causes a player to in most cases take significantly less damage and allows them the chance to perform a new action sooner than if they were to get hit without blocking. Specific attacks can be performed with motion inputs (pressing a series of movement buttons) followed by an action button. For instance, the"quarter circle forward" motion formed by first pressing the down key, then the down and the forwards key, and lastly the forwards key in that order is commonly used as an activation method for attacks.

Fighting games are often built on variations of the deterministic lockstep system [6]. This system entails that a game instance will wait for specific time intervals, for instance 1/60th of a second when running at 60 FPS, before generating the next gamestate using only the previous gamestate and the inputs provided to the game engine at that interval. Additionally, all calculations performed in generating a gamestate must be deterministic. By enforcing this, all connected game instances should always generate the same gamestate when given the same inputs.

In the context of fighting games, the term frame can be related to two different concepts. Regarding games in general, a frame is a single image generated by the game and displayed on a screen. A game engine generates multiple of these per second to communicate the state of the game to the player, and as such frames are also often used as abstract representations of time. For instance, when a game runs at 60 FPS, one frame takes roughtly 16.666... miliseconds. In terms of fighting games, the word frame can also be referring to frame data. Fighting games often have a variety of characters to choose from, and each of these characters have their own unique actions they can perform. Frame data refers to the data that encodes each of these actions. For instance, the frame data of an action can describe how long it takes before the hitboxes of a move come into play or how long a character has to wait after using a certain action to perform other actions again.

Concerning actions, modern fighting games include an action buffer for each player. This buffer stores a player's input for a certain amount of time after a player has performed it, this period can be referred to as an input lifetime. When interpreting a player's inputs and translating them to the corresponding action, the oldest inputs in this buffer will be referenced first. This is mostly done so that players have to be less precise when performing multiple actions consecutively. Before input buffers were introduced, any inputs a player performed before their current action had ended would be discarded immediately, making it hard to effectively link multiple actions.

In case an action causes two players to take up the same space (for instance if one player jumps into another player), a collision occurs that will have to be resolved. This often involves pushing the players away from each other following a certain rule. In order to stay deterministic, this rule will have to be consistent in its application between the two players. Another form of collision can occur between two character's hitboxes and hurtboxes. A character's hurtbox is the area where they can be hit by an opponent's action, and a character's hitbox is the area with which a character's action can hit an opponent.

In terms of network communication fighting games tend to use either delay-based netcode, rollback netcode or a hybrid of those two netcodes. Delay-based netcodes artificially delay the time between when an action is performed by a player and when a gamestate is generated using the input of that action to create time for a player's inputs to be received by the opponent. For instance, if a game has 10 frames of delay, the player's input will only be processed 10 frames after it has been performed and in those 10 frames the input is sent to the opponent so it can be processed there to generate the same frame. In case an input is not received in time by the opponent, both game instances will freeze and wait until all inputs have been received to ensure both game instances remain equal.

As freezing the game is quite intrusive to the playing experience, rollback netcode was developed as an alternative. Instead of freezing a game when the opponent's inputs have not been received yet, rollback netcode makes a prediction of what the inputs would have been and generates the next frame using that prediction. When such a prediction is made, the gamestate before the prediction is stored internally in case the prediction ends up being wrong. When the message for which a prediction was performed is received, the prediction will be compared to it. In case this prediction was wrong, all gamestates including the wrong prediction will be deleted (rolled back) and the correct game state will be generated retroactively using the received input. Commonly, no more than 7 frames at a time will be stored. This is because needing to re-generate more than 7 frames in case it takes long for a message to come through can cause the gamestate generation to take longer than the interval it has to be generated in. If this 7 frame limit is reached, the game will be freezed until all messages have been received and processed.

Most modern fighting games use a combination of these two strategies. In effect, this means that fighting games do not have to freeze any game instances as long as messages are received within the amount of delay frames plus 7 rollback frames. The amount of delay frames tends to be 5 as any more than 5 frames starts to noticeably impact the playing experience. Altogether, this means that modern fighting games can commonly miss a message for up to 13 frames before a freeze is performed.

2.2 Definitions

The topic of netcodes in the context of video games has not seen a lot of attention in academic literature. As a result, the terminology around the subject has yet to solidify into distinct and seperable terms. This is especially true for the term netcode itself. Per example, in *"Analysis of Netcode, Latency, and Packet-loss in Online Multiplayer Games"*, Ahmed et al. start their introduction with the following definition:

> ""Netcode" ... is a technical term that is used to define how good or bad a game's network connection is and how well the network is synchronized between clients and servers." [7]

This definition uses the term netcode to describe the quality of a game's connection and synchronization between connected systems. Meanwhile, in *"An analysis of continuous consistency models in real time peer-to-peer fighting games"*, Martin Huynh, Fernando Valarino et al. start their introduction with this definition:

> "Netcode is a term often used haphazardly within the video game community to describe the overall implementation of networking in a video game." [8]

In contrast to the previous interpretation, this definition uses the term netcode to describe the complete implementation of networking in a video game rather than the quality of said implementation. Relevant as well, in *"Improving Input Prediction in Online Fighting Games"*, Anton Ehlert provides the following definition of the term netcode in the Rollback netcode and replay data section:

> "Netcode is a colloquial umbrella term that refers to the technical implementation of realtime networking and state synchronization in online video games." [9]

This argues that netcode refers to the realtime networking and state synchronization implementation of a game rather than the overall implementation of networking. Different still, in *"Polka: A Differentiated Deployment System for Online and Streamed Games, Meta-verses, and Modifiable Virtual Environments"*, Jerrit Eickhoff et al. introduce the term netcode in the section Online Game Networking as follows:

> "... is heavily influenced by online game networking, referred to as netcode." [10]

This depicts netcode simply as a different term for game networking. While all of these definitions refer to (the quality of) a game's communication and synchronization strategy, they do not paint a clear and consistent picture of what the term netcode means exactly. Further complicating this is the existence of the terms delay-based netcode and rollback netcode, both of which describe networking strategies, with the latter in particular concerning itself more so with how gamestates are handled rather than the act of networking itself [4]. Given the conflated state of this terminology, we propose three terms which together allow for clear and comprehensible communication in regards to the different aspects of video game networking throughout this thesis.

Firstly, we propose the term statecode, defined as follows:

Definition 2.1 (Statecode)**.** The implementation module of an online multiplayer game responsible for upholding the canonical state of a game instance.

Using this term, we can more accurately describe delay-based netcode and rollback netcode as delay-based statecode and rollback statecode respectively. This clarifies that the strategies they define are more closely related to the way a game's logic handles the preservation of its canonicity rather than the act of networking itself.

Secondly, we propose a new definition for the term netcode:

Definition 2.2 (Netcode)**.** The implementation module of an online multiplayer game that is responsible for starting, upholding and managing network connections, as well as sending and receiving network packages.

This term distinguishes itself from the term statecode by more closely focusing on the act of networking. In this way the term is isolated from any game logic, clarifying that it solely refers to a game's networking strategy.

Lastly, we propose the term net-state code, which combines the two previously discussed terms. It is defined as follows:

Definition 2.3 (Net-State Code)**.** The combination of a statecode and netcode which together represents the game's communication pattern as a distributed system.

With this definition in mind, we can attribute the previously quoted term defined in *"An analysis of continuous consistency models in real time peer-to-peer fighting games"* and *"Improving Input Prediction in Online Fighting Games"* as net-state code rather than netcode, thus creating a clear distinction between an online game's full communication pattern and its networking strategy.

Additionally, we give a definition of accuracy in the context of gamestates. In a deterministic game engine, the same series of inputs should always result in the same gamestate being generated. Given this, a correct gamestate should match the gamestate generated by a game given the same inputs running without any networking[2]. Accuracy could then be described as the ratio of states that are equivalent to such a local emulation given the full series of inputs, or the State Equivalence Ratio. Mathematically, we can define the State Equivalence Ratio as:

Definition 2.4 (State Equivalence Ratio)**.**

$$SER = \frac{S_{equiv}}{S_{total}} * 100\%$$

Where SER is the percentage of game states equal to a local emulation, S_{equiv} is the number of game states equal to a local emulation and S_{total} is the total number of game states generated.

[2]Some game engines generate seeds to use for random number generation during a session or match whilst maintaining determinism. In such cases, the seed would have to be the same between game runs as well to recreate the same state.

3 Related Work

In this section, we discuss work that is related to this thesis.

Recently, Elena Stroiu created a benchmark to test games made in the Unity engine[3] on their netcode implementation, focussing on Minecraft-like games with high player counts and input throughput [11]. This provides a valuable tool to game developers wanting to find the proper netcode to use for their Unity projects, but this leaves many other platforms and use-cases unsupported. This thesis hopes to meaningfully expand the selection of benchmarks that can be used for testing purposes.

Similarly, Raymond Leonardo Chandra et al. tested a set of netcodes in the Unity engine on various performance metrics such as RTT and FPS in VR games [12]. While the conclusions drawn may be quite detailed and valuable to Unity VR developers currently, we aim to create a benchmark that can allow developers or researchers to test any number of netcodes without relying on academic work for relevant results.

More specific to fighting game research, Anton Ehlert created and tested a different prediction agent to use for rollback statecode input prediction rather than the simplistic repeatlastframe policy [9]. This approach focuses on improving rollback statecode directly whereas this thesis aims to test broadly available netcodes on how well they perform in tandem with a game's statecode.

Related to statecode implementations, Martin Huyn and Fernando Valarino performed an experiment to test the user experience of playing a fighting game with either a delay-based statecode or rollback statecode in various network conditions [8]. This work provides valuable insight into how specific network conditions can impact a player's experience, but only provides one static netcode implementation to test the user experience with. While we have performed experiments under the same network conditions, we hope to create a more broadly useable benchmark to give further insight into how specific netcode implementations can impact the networking performance of a game under various network conditions.

Lastly, Ricky Pusch wrote an article that has served as a valuable and extensive resource for understanding how delay-based statecode and rollback statecode work and differ from each other [13]. This article was published right around when the previously proprietary GGPO rollback statecode was made available for public use in October of 2019. GGPO, developed in 2015, was the first rollback statecode. It was not until its release to the public that rollback statecode became more widely adopted. Because the article was released around this crucial time and contained an exhaustive yet understandable explanation of how both statecode approaches worked it has become a widely known resource for both fighting game players and statecode research.

[3] https://unity.com

4 System Design

In this Section we describe the design of the system introduced by this thesis. The system consists of two components. The first component is Benchy-Fighters[4], which is a generic multiplayer fighting game which allows its netcode to be interchanged. The second component is ButtonCheck[5], which is a benchmark with modifiable network conditions. We will define the requirements for both of these systems, explain their general implementation and highlight specific implementation details.

4.1 Benchy-Fighters

In order to fairly assess the accuracy of various netcode implementations against each other, we need a consistent statecode implementation to test each netcode against. To ensure each netcode is treated equally, we created a generic multiplayer fighting game called Benchy-Fighters wherein the netcode used can be changed before each game run.

4.1.1 Requirements

R1 *Complexity:* The game's complexity should match that of an actual fighting game. To this end, the game will need to perform all calculations needed for determining the player's actions, moving the player on the screen, and calculating a player's health and movement trajectory after they have been hit. Benchy-Fighters' complexity ensures that even if its full workload is lighter than that of an actual fighting game, it does fairly represent how the computation time of the game scales depending on the inputs provided in comparison to the computation time of the netcode used. To implement this, we will have to take into account all basic functionality found across most fighting games and model these accurately.

R2 *Interchangeability:* The game's net-state code must allow for netcodes to be interchanged freely and easily without this impacting how the statecode interacts with the netcode. In order to achieve this, all points of interaction that the game has with its netcode will need to be generalized to allow any netcode implementation to be used by the program. Benchy-Fighters' interchangeable netcode implementation means that it can be expanded to use any netcode that matches the environments it can run on. In order to allow this functionality, we will have to define a generalized netcode structure that can contain a wide variety of netcode implementations.

R3 *Fairness:* The game must treat each netcode equally. To ensure this, the game will need to keep deviations from its generalized netcode structure to an absolute minimum and never make decisions based on the netcode in use after configuration. By treating each netcode fairly, Benchy-Fighters does not inherently favour any type of netcode implementation over another and can thus accurately show the benefits and shortcomings of each individual netcode implementation.

[4] https://github.com/michiel9797/Benchy-Fighters
[5] https://github.com/michiel9797/ButtonCheck

R4 *Configurability:* The game should allow various configurations to be applied so it can be used in various test environments. Variables such as if the game instance should act as a server for other instances to connect to, the IP address of the game instance, which IP address the game should connect to, what input data it needs to use and what netcode it should use need to be easily interchangeable between each run of the game. With its flexible configurability, Benchy-Fighters can easily be used in various test environments without requiring its code to be extended.

4.1.2 General Implementation

Since the ability to switch the netcode being used without infuencing the rest of the program execution is a key design factor covered by both R2 and R3, using a programming language that supports switching implementations is key. To this end, Benchy-Fighters was written in C++. By making use of object-oriented programming, namely abstract base classes and polymorphism, Benchy-Fighters can allow each netcode implementation to form its own class and implementation without changing the way the rest of the program interacts with the netcode implementation.

A lot of different functionality is required to run a match of a multiplayer fighting game. To this end, and in service of R4, Benchy-Fighters is built with three discernible execution modes: emulation, server simulation and client simulation. We will discuss all three modes in order of their creation, explaining important implementation details as they become relevant.

Firstly, the emulation mode was constructed. In order to judge the accuracy of any given frame generated, a correct frameset has to be generated ahead of time. The emulation mode achieves this by running the game engine in a single program instance without using any networking capabilites. In emulation mode, Benchy-Fighters will emulate a two player match by directly feeding a game instance the input files from both players, thus sidestepping the need for any networking and in the process giving us the computation time for running a match with the given input without networking.

In accordance with R1, the overall game implementation focuses largely on creating a generic lightweight system that matches the complexity of an actual fighting game. To this end, functionality like a user interface and realtime inputs were not implemented. The systems of Benchy-Fighters are loosely based on those present in "GUILTY GEAR -STRIVE-", though stripped down to the most basic functionalities [14]. The only playable character implemented was based on the frame data of Ky Kiske as provided by Dustloop [6].

Secondly, the server simulation mode was constructed. Most of the netcodes reviewed for implementation worked only with a client-server model instead of a peer-to-peer model, so the server simulation mode was implemented to house this communication. The server implementation is rather simplistic, only serving to send a signal to start a match once both game instances have connected to it and passing messages between game instances. When the server instance detects that one of the game instances has disconnected after the match has started, it will shut down regardless of if the other game instance is still connected. The server runs using the same time

[6] https://www.dustloop.com/w/GGST/Ky_Kiske

intervals as the other execution modes since the netcodes implemented, and most netcodes reviewed, rely on receiving at least one message every 16.666... milliseconds to uphold the connection.

Lastly, the client simulation mode was constructed. Due to the size and complexity of most of the components introduced in this mode, we have subdivided these components into their own subsections. However, there are a few notable details worth discussing beforehand. One of these is that the only point at which the netcodes are treated differently by the main program is as the clients are initialized. This process requires the constructor of each specific netcode implementation to be called, and as such cannot be generalized without rolling all implementations into one object which would negatively impact R2. Another noteworthy detail is that all netcodes first have to establish a connection to the server with a handshake before they are able to start sending and receiving messages. As this process differs for each netcode, a function was assigned to house this process before the main game loop starts. Only once this function returns true will the main game loop be accessed and will the client begin waiting to receive the start of match signal. Lastly it is also important to note that the statecode only accepts reliable in-order communication. While a custom implementation of an unreliable unordered strategy may have been more efficient, we chose reliable in-order due to time constraints as the statecode implementation would have to be expanded upon greatly in order to support out-of-order communication. Additionally, it is not uncommon for multiplayer games, including fighting games, to not be optimized in this regard.

Also of note, as Benchy-Fighters was made for the purpose of benchmarking netcodes, instead of reading live input from players it loads in all inputs from one or both players (depending if we are running an emulation or simulation repsectively) beforehand from an input file. By using input traces where each input has been stamped with the frame they were performed in, the same game can be simulated multiple times. This way we can avoid the noise that differing input traces can cause. This does require users to either generate their own input traces to be used with Benchy-Fighters, or use one of the input traces provided with the program.

4.1.3 Engine Implementation

The implementation of the game engine follows a lot of the concepts outlined in Section 2.1, namely the fact that Benchy-Fighters uses a Deterministic Lockstep system. Any problems that could lead to loss of determinism, such as how to resolve a collision, are resolved by reducing the problem to a binary question. For instance, in case of collisions, this is done by always moving the player that is closest to the right side of the screen out of the player on the left side of the screen.

Generating a new gamestate is done in six steps, each step concerning itself with a different section of game logic:

1. Manage player inputs
2. Tick up the current frame
3. Tick the player's movements
4. Check for active hitboxes
5. Apply any possible hit effects
6. Set the next player actions

Important to note here is that the results of a player's actions are only calculated the frame after they have been determined. This is because under certain circumstances a player's actions may be disregarded for a frame, and generating gamestates in this order allows checking if a player's actions should be ignored to take place within a single frame instead of necessitating a way by which that information can be carried to the next frame.

The first step, managing player inputs, is unique in the fact that it is not performed by the game engine itself. Rather, the game engine features a set of public functions that allow the part of the program responsible for receiving messages to distribute the inputs received to the correct players by loading these inputs into each player's input stack. After all inputs have been loaded into the stack for each player, a public function is called to manage each player's stack and input buffer. This function removes any inputs from a players input buffer when their lifetimes have ended and moves any inputs from the stack into the buffer when the frame matching the frame during which the input was performed is generated[7].

The second step, ticking up the current frame, progresses any action players were taking in the previous frame up to the current frame. In case a player's previous action has ended due to this, the player's inputs will be considered in a later step to determine their next action.

The third step, ticking the player's movements, looks at all forces that are being applied to a player and moves them in accordance to them. Aside from a few specific exceptions, this step is the only step during which the location of a player changes. This is also the step where collisions between players are resolved and the direction players are facing is changed if they swapped sides (if a different player became the leftmost player on the field).

The fourth step, checking for active hitboxes, looks if any player is performing an action that in its current frame has a hitbox attached to it. If this is the case, these hitboxes will be checked against the other character's hurtboxes. This is done for both characters before any of the consequences of being hit are applied, so it is possible for two players to hit each other on the same frame.

The fifth step, applying possible hit effects, changes the state of both players in accordance to if they were hit or not. In case a player was hit, the game checks if that player was blocking. In this case, the blocking player is immediately pushed back slightly and does not receive damage. This also causes the player's inputs to not be considered when determining their next action this turn. If a player was hit without blocking, the properties of the move they were hit by are applied to them, and they enter a state where they cannot perform any new actions until they hit the ground. In case one player hits the other player without being hit themselves, their action is ended early so they can perform another action the next frame.

The sixth step, setting the player's next action, looks through the inputs in each player's input buffer and determines what action they correspond to. The oldest action button pressed always takes priority in determining a player's action. In case a player's input buffer contains an action button, the game first checks if there is a move in the corresponding character's action list that matches the motion inputs performed by a player. If there is, that attack is selected as the next

[7]Due to the implementation of the delay system, certain inputs can be received before they need to be processed. These will remain on the stack until the engine generates the gamestate for which the input is needed

action. If there isn't, the attack corresponding to just the action button is selected instead. If no action button is present, the movement buttons will be checked. In case a player wants to jump, the forces applied to them will be changed so the jump is performed in the next frame during the second step. In case a player wants to walk left or right, this movement is performed immediately.

Lastly, the game checks if any player has reached 0 health this frame. If that is the case, the fact that the game has ended will be recorded in the gamestate. After these steps have been performed, a public function will be called to print the gamestate out to the command line.

4.1.4 Statecode Implementation

To explain the statecode implementation, we will continuously be referring to Figure 1. This figure shows a simplified flowchart that depicts the main game loop and its rollback statecode implementation. It should be noted that the real system uses a combination of rollback statecode and delay-based statecode, though for the sake of readability the second part has been left out of the flowchart among other details. We will be discussing each code locality, broadly depicted by each layer of the flowchart, and highlighting both what part of the rollback statecode it implements and any noteworthy details not included in the chart.

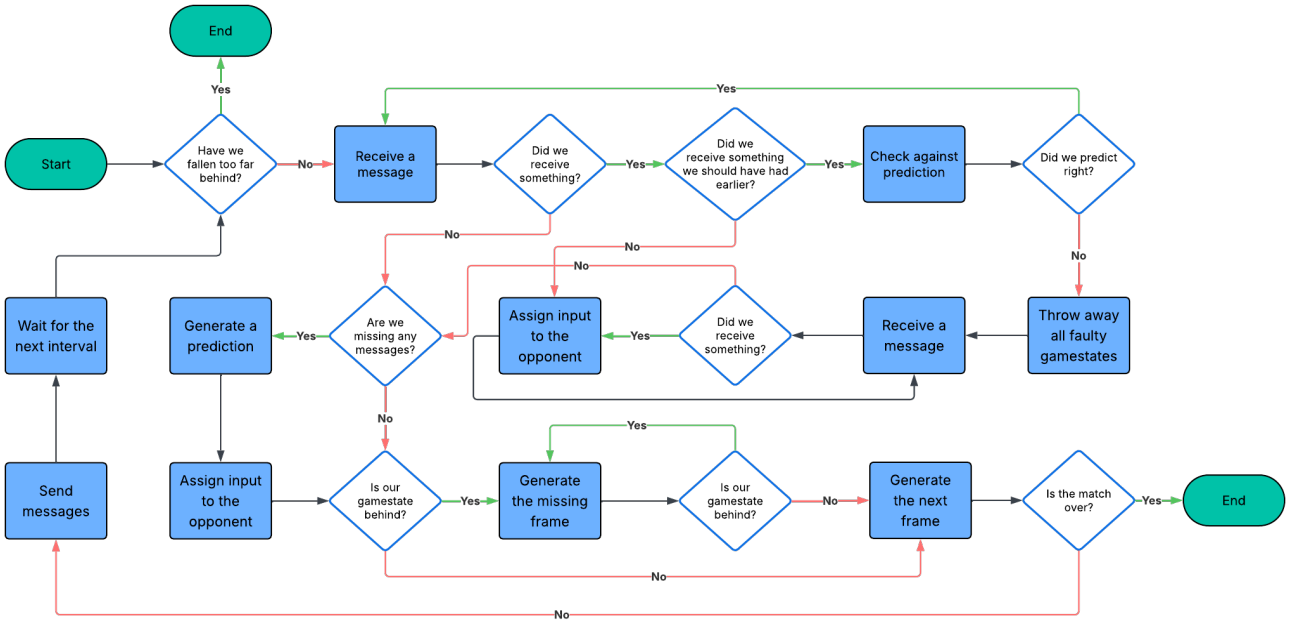

Figure 1: A simplified flowchart depicting the main game loop

Before the first code locality, we check to see if we have not received communications from the opponent for 12 frames consecutively (at 60 FPS this gives $12 * 16.666... = 200ms$). This margin consists out of 5 frames of delay and 7 frames of rollback, and specifies when a gamestate is too far behind to be recoverable. Rollback statecodes only roll back a maximum of 7 frames as rolling back any further may result in the game not being able to resimulate the rolled back frames in time for the next frame to be generated. We will go further into the 5 frames of delay when explaining the

third code locality. Also of note is that the code that waits for the start signal occurs before this chart, effectively meaning that the start of this chart coincides with receiving the start signal.

The first code locality shown on the upper layer of Figure 1 depicts the prediction comparison segment of the rollback statecode implementation. This section loops over any messages that are received too late (i.e. after the frame that required it has already been generated) and compares them to the input we had predicted for that frame. Since we use a standard repeatlastframe policy[8], we can simply compare each message received to the message we used to predict the previous frame(s). If these match, we can keep the frame in which we used that prediction and can move on to check the next message. If these do not match, we have to roll back all frames up to and including the frame in which that prediction was made to ensure the gamestate does not diverge.

The second code locality shown on the middle layer of Figure 1 depicts the input assigning step. This section loops over any messages received and assigns them to the opponent. Since we can only reach this step with our gamestate either at the message we are about to receive or rolled back to the message we are about to receive we can directly assign any inputs received to the opponent. Since we use the reliable ordered message structure, once we start receiving messages we should receive all messages up to and including the message we need for this frame. This means that we can only miss the message we need for this frame if we did not receive any messages at all. In this case, we generate a prediction using the repeatlastframe policy and assign this prediction to the opponent.

The third code locality shown on the bottom layer of Figure 1 depicts the frame generation and resimulation segment of the rollback statecode implementation. The only circumstance in which our gamestate can be behind is if earlier on we rolled back our gamestate due to a faulty prediction. In this case, we resimulate every frame that we are behind using the correct inputs we received during the previous step. Afterwards we generate the frame we are currently on and check if the match has ended in this frame. If it has, we shut down this client. If it has not, we send any messages we need to send and wait for the next interval to repeat this process. To implement the delay-based statecode into the actual program, the system waits the first 5 frames instead of generating the next frame. In doing this, the gamestate will always be 5 frames behind the inputs we send to the opponent, meaning that each message sent can be up to 5 frames too late before we would have to do a prediction instead.

As this statecode implementation only interacts with the netcode by receiving messages or sending messages, both interactions that have been generalized in the netcode implementation, the statecode implementation upholds R3.

[8]This policy states that we predict the input of the current frame to be equal to the input of the previous frame. When predictions stack, all predictions will be equal to the last input received.

4.1.5 Netcode Implementation

To explain the netcode implementation, we will be referring to Figure 2. This figure shows the abstract base classes used by Benchy-Fighters and an example implementation of a netcode using those abstract base classes.

In order to uphold R2, we decided to implement the interchangeability of the different netcodes by way of an abstract base class. In doing so we can ensure that as much code will be reused between netcode implementations as possible. We decided to extend this to include an abstract message class as well in order to more closely follow the general implementations of the netcodes implemented, and in doing so reduce the overhead other implementations would incur.

We split the client and server objects into separate classes to avoid confusion on which one is being used where and to ensure the methods for each class are optimized for its specific use case rather then generalized to work for all cases. This also makes it easier to implement new netcodes, as most netcodes also feature a client object different from the server object.

Through implementing the netcodes using an abstract base class, Benchy-Fighters can also be easily extended to include new netcodes. Outside of the exception regarding the construction of each class mentioned in Section 4.1.2, all a developer would have to do to implement a new netcode into Benchy-Fighters is implement a class based on the NetworkLayer, ClientLayer and ServerLayer abstract classes. The required functionality of each function is explained thoroughly in the project files, and the three already implemented netcodes serve as examples on how to integrate various netcode structures into the abstract base class.

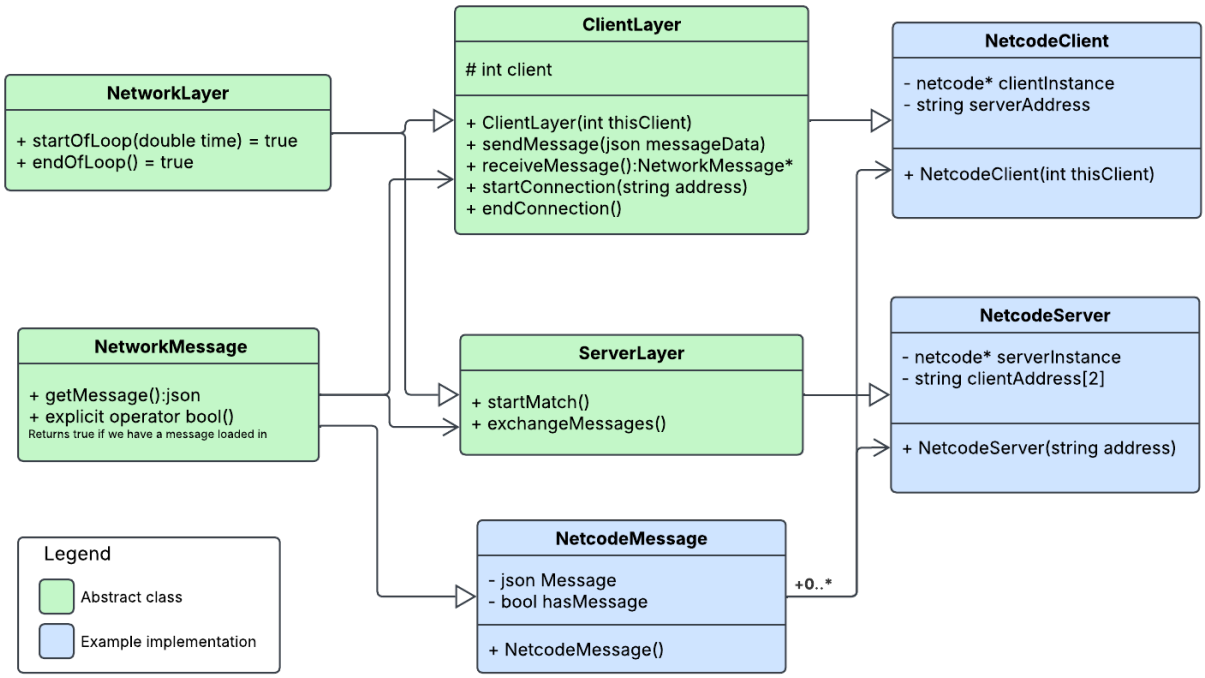

Figure 2: An example of a netcode implementation using the abstract base classes

4.2 ButtonCheck

In order to benchmark the netcodes implemented within Benchy-Fighters, we produced a benchmark called ButtonCheck that creates a network environment with modifiable network conditions to test multiplayer games on their accuracy and networking CPU time.

4.2.1 Requirements

R1 *Ease of use:* The benchmark should be simple to understand so that developers can use it to test netcode implementations without confusion. The user interface will need to be clear in its design and layout, and its use-cases should be well documented and communicated. Through its ease of use, ButtonCheck provides a testing solution to a broad variety of developers. This requirement can clash with the needs for functionality and configurability, so a consistent design approach and clear documentation will be required to balance this out.

R2 *Relevance:* The benchmark must be configurable in ways relevant to the testing goals and the results must be relevant to the benchmark's use-case. We will need to study the various requirements developers decide on when choosing a netcode and ensure the benchmark provides the most relevant metrics needed to make this decision. These include low networking overhead and good accuracy under network stress. By providing developers the CPU time and accuracy recorded during benchmarks, ButtonCheck can be a valuable tool for improving and streamlining the netcode selection process.

R3 *Clarity:* The benchmark should present its metrics in a straightforward and easy to understand manner. To this end, the performance metrics displayed will need to be scrutinized and modified to represent the results clearly while not convoluting their meanings. By presenting its results clearly, ButtonCheck will enable developers to make well-informed decisions based on a given netcodes advantages and disadvantages.

R4 *Configurability:* The benchmark must provide a wide array of options for the user to configure the program to their needs. Various network conditions including latency and packet loss, should be editable to create a testing suite. In this suite, the user should be able to decide exactly when the latency and packet loss should be changed, and to what values they should be changed. The user should also be free to change both the underlying game instance, the information given to this instance and the input data provided to this instance. With its flexible configurability, Buttoncheck can provide a varied testing environment that can assist developers in testing netcodes in specific conditions.

4.2.2 General Implementation

In order to abide by R4, ButtonCheck was designed from the ground up to provide the user with a variety of ways to customize a benchmark run. In order to properly cover its various functionalities, we will explain both ways in which the benchmark can be operated and highlight any relevant settings.

ButtonChecks' primary interface can be accessed by calling it from the command line without any arguments. Doing so, the menu shown in Figure 3 will appear. Outlined here are the most relevant settings to running the benchmark: the paths to the game client and server, the paths to the inputs to use and the path to a network emulation file.

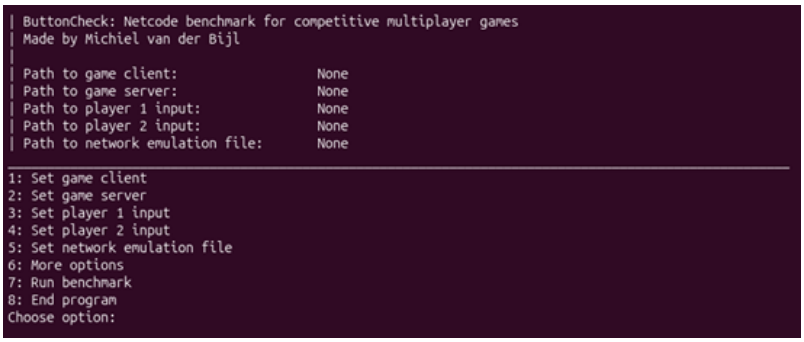

Figure 3: The main menu of ButtonCheck

In order to allow users to define their own network conditions under which to perform tests, ButtonCheck comes with its own network emulation engine that can read a given network emulation file and apply its specifications to a benchmark run. The engine uses TC Netem calls from the Linux shell to apply network conditions to the test environments it sets up. When these conditions are applied is based on the number of frames the client program has produced so far. For instance, a user can specify that they want the program to apply 50ms of packet loss from frame 60 onwards. We chose this method over having the network engine time when to apply the network conditions internally because we believe it allows ButtonCheck to apply its network emulation as consistently as possible without making any broad assumptions about the client program. An example of a network emulation file can be found in Appendix C.

Choosing 'Run benchmark' will start the benchmarking process. This process consists of roughly three steps, all of which are depicted in Figure 4.

The first step is the emulation step, during which the output and CPU time of a single game instance are recorded. The game instance used corresponds to the program stored at the game client path as given by the user. Aside from a handshake performed before the game starts, all standard output data the game program produces is stored in a file to be used for comparison later on. This method was chosen as output values for game states can become very large depending on the complexity of the game, so while this approach sacrifices the speed at which the final comparison can be performed, it does allow for games of any gamestate complexity to be supported.

The second step is the simulation step, during which the output and CPU time of a game instance that is playing over an emulated network connection is recorded. To ensure the connection has been set up correctly, a server instance is launched first. Only after this server has successfully performed a handshake with the benchmark program will the two client instances be launched. The benchmark receives the output of one of these client instances and stores it in a file.

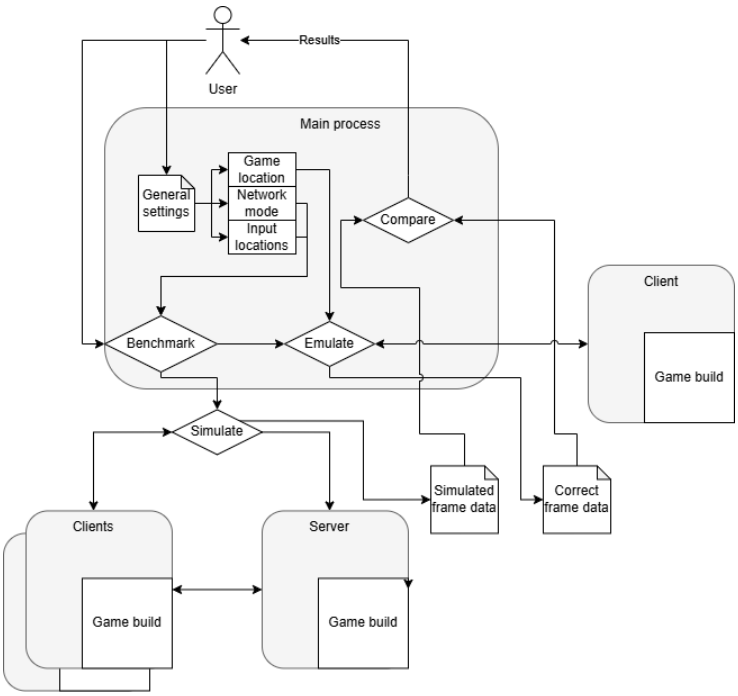

Figure 4: A diagram of ButtonCheck

The last step is the comparison step. This step is only performed if files for both the simulation and emulation steps are present. During this step, the benchmark will compare every line in both files to see if they differ or not. When calculating the accuracy as a percentage, that percentage will be based on the number of correct frames relative to the highest amount of frames generated between the two runs. Effectively, this means that a simulation run is punished for generating more frames than the emulation run as each additional frame is counted as incorrect towards the accuracy percentage. In case both the emulation and simulation steps were performed, the benchmark also reports the difference in execution time of the recorded client program between the emulation and simulation steps. This format was chosen over reporting both execution times seperately to ensure that the number reported is not influenced as heavily by the amount of resources that are dedicated to the benchmark and the programs it calls, and to provide the user with a more easily understandable metric conform with R3.

Choosing 'More options' in the menu depicted in Figure 3 instead will present the user with the menu shown in Figure 5. This menu primarily provides smaller, situationally less relevant options that allow a user to streamline their test runs or adjust how the benchmark should behave. The first two options allows the user to skip the emulation and simulation step in the benchmarking process in case these steps are not required for the purpose of a test. The second set of options allows a user to save the output data provided to the benchmark to their system, which allows a user to look through the output themselves to find possible issues and confirm the benchmarks results. Notably, if a user saves the emulation/simulation file and skips the emulation/simulation step the previously saved file(s) will be used in the comparison step, saving time at the cost of the CPU times not being measured.

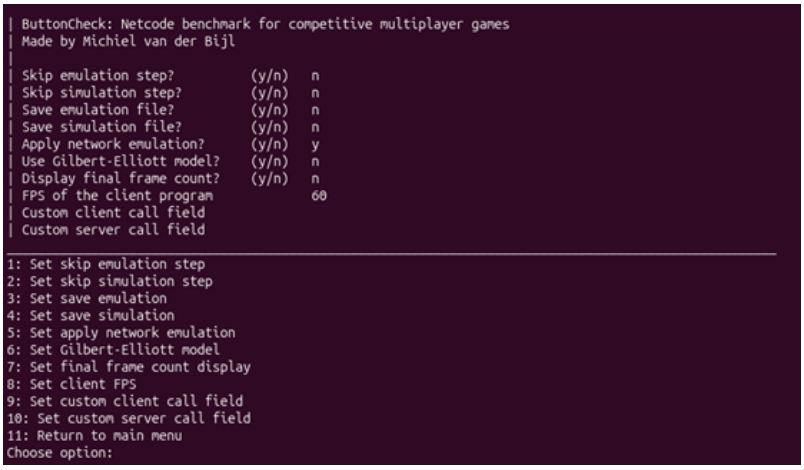

Figure 5: The extra options of ButtonCheck

The fifth option allows the user to turn off the network emulation during the simulation step. This option stops the benchmark from reading the network emulation file and applying its contents, ensuring that the simulation step will be performed with perfect network conditions. The sixth option allows the user to use the Gilbert-Elliott model for interpreting their network emulation file. This model gives the user better control over how bursty they want the packet loss applied to be, which more accurately mirrors real word network conditions.

The seventh option allows for displaying the total amount of messages the benchmark has received from the emulation and the simulation run. The eight option allows a user to set the FPS at which the game they supplied to the benchmark runs. The function of this option is purely cosmetic, as the benchmark does not control the client's FPS. Instead, this changes the amount of frames the benchmark needs to receive before it increments the match duration timer it shows the user during testing by one second. The match duration timer was implemented in accordance with R1, giving the user visual feedback that the program is still running during the benchmark.

Lastly, the ninth and tenth options allow a user to pass a parameter directly to the client and server programs when they are launched during the simulation step. This can help developers who may be interested in adjusting their games to be used for ButtonCheck, as it gives them an extra interface through which to provide their programs with parameters not covered by the benchmark itself.

In addition to its primary interface, ButtonCheck can also be called from the command line with arguments to run multiple tests in a batch. When running the program from its primary interface, all settings that have been changed are saved to a file, wich also functions as a config file when running from the command line. When the program is called with the arguments -F X (X being any positive integer), the program will perform X benchmarks using the settings saved in the config file. This allows users to set up and test a benchmark using the primary interface, and then perform a batch of benchmarks with the specified settings using a command line call.

4.2.3 Network Structure

During the simulation step, ButtonCheck sets up a network structure as depicted in Figure 6.

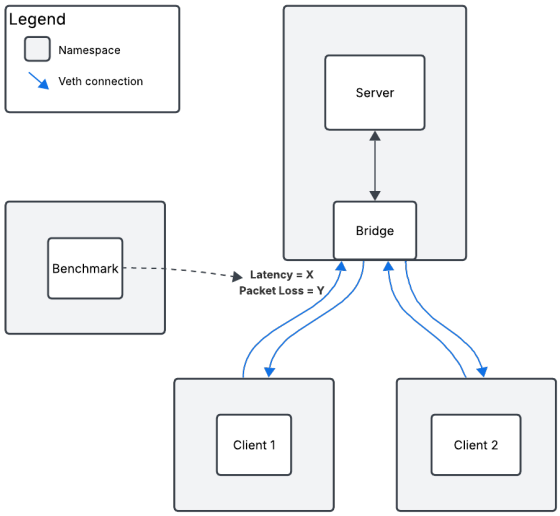

Figure 6: A diagram of the network structure

First, the benchmark creates unique network namespaces for both of the clients and the server. A network namespace is a Linux feature that allow any programs within such a namespace to use an isolated virtual network stack. Essentially, the Linux kernel treats these namespaces as unique isolated devices on a network and handles their networking separately from the rest of the device. By running each program in its own network namespace, we can ensure they cannot communicate with each other in any way other then the means provided by ButtonCheck. Then, it creates two veth pairs. Veth pairs, virtual ethernet devices, act as a tunnel between namespaces. It has two ends, allowing it to connect two namespaces. The benchmark moves one end of each veth pair into the namespaces of the clients and the other end into the namespace of the server.

In order to ensure that both client programs can send their messages to the same IP address and the server can receive the messages from both clients cleanly, the benchmark creates a bridge for both of the server side veth interfaces to connect to. Then, it assign each of the client side veth interfaces and the bridge a unique IP address for the client instances and the server instance to use. Subsequently, it applies network conditions to one of the veth pairs using TC Netem. The network conditions include latency, packet loss, and in case the Gilbert-Elliott model is used how packet loss burst patterns should be formed.

Of note is that the benchmark specifically applies the network conditions only to the connection from the first client to the server, while it uses the second client's output and CPU time for the results of the benchmark. We chose this approach mainly to avoid any desynchronization issues that could occur from applying the network conditions to the messages sent by the server. Primarily, we want to ensure that the message to start the match is received by both clients at the same time. As this research is mainly interested in the accuracy of the tested netcodes after a match has been started, we have not implemented any techniques to ensure both client instances start the match at the same time even if network conditions interfere with receiving a message. As such, if one of the messages to start a match would be dropped and resent before it reaches a client, said client would start the match multiple frames behind the other client with no way to catch up.

4.3 Design Limitations

Before we describe the experiments performed, we would like to touch upon a big limitation of Benchy-Fighters. We are aware of the existence of a bug in Benchy-Fighters that sometimes causes a sleep between lockstep cycles when simulating to last far longer than intended. Thorough test have been performed, but as of yet no pattern has been found in when this bug occurs. In order to perform the tests without too much interference from this bug, Benchy-Fighters sends out an error message each time the bug occurs specifying how long the sleep ended up being. Each test where this error message was sent was discarded and repeated directly after the initial testing. While this does mean that all the results gathered are based on tests performed without the bug occurring, it is possible that this bug caused tests that went on for longer to be discarded more often than shorter tests. It should also be noted that this bug occurs both when testing using WSL and through a Linux virtual machine. It has yet to be tested if the bug occurs in a pure Linux system. For the sake of transparency, Tables 7 through 10 show the total amount of tests performed per datapoint and the percentage of those tests that was discarded due to the occurrence of the bug.

5 Experiments

In this section, we use the developed benchmark and game to evaluate three netcodes: Yojimbo, GameNetworkingSockets and RakNet. We will shortly discuss how we gathered the input data used and how we selected the netcodes to be implemented, and then we will perform three experiments on the selected netcodes to study how they compare.

5.1 Input Data

The input data used in each experiment was gathered through a play session of the game Benchy-Fighters was built to resemble, "GUILTY GEAR -STRIVE-". During this play session, two experienced players chose Ky Kiske, the template for the only character in Benchy-Fighters. Additionally, the players only performed actions that exist within Benchy-Fighters. The match was restarted in case an illegal action was performed. They also played on the same local network to ensure neither player experienced any network issues.

After a legal match had been played, the actions performed were annotated to a JSON file by playing back the replay data of the match frame by frame and manually noting down each button press when it was performed. In order to ensure the lifetimes of each button press matched that of the match played originally, certain inputs were repeated in such a way that they would stay in the action buffer for the full duration it was observed to be in the action buffer during the original match.

One full round of input data from both players was gathered and used across all experiments. Differences in input data could result in different workloads to process the input and some series of inputs may be more prone to needing resimulation then others. For instance, if a player stands still for three seconds, no resimulation would be needed if a message was dropped for that period. Whereas if a player performs multiple actions in that time, it is more likely that an input leading to an action is dropped and would need to be resimulated later. In order to avoid such noise that differences in input patterns could cause, we decided to only experiment with one set of input data from both players.

5.2 Netcode Selection

In order to select a set of netcodes for testing, an extensive search was performed to find netcodes that matched a set of criteria. After this selection process had been completed, a thorough online source of open source netcodes, sources regarding such netcodes and various tools for network simulation was discovered: multiplayernetworking.com [5]. All netcodes that were under consideration are also listed there.

Netcodes were chosen based on the following four criteria:

1. Is the source code of the netcode open for public use? In order to implement a netcode, its source code would be required for it to be able to be implemented in Benchy-Fighters. Only netcodes with their source code publicly available were considered.

2. Can the netcode be compiled in Linux? Since TC Netem, the chosen network emulation tool, is exclusive to Linux, only netcodes that can be compiled in Linux were considered.

3. Does the netcode support Ordered Reliable network communication? Using different communication methods between netcodes could inherently favour one netcode over another. While a custom Unordered Unreliable communication implementation would have been more efficient, in order to narrow down the scope of the work, only netcodes that support Ordered Reliable communication were considered.

4. Is the netcode written in C or C++? Benchy-Fighters was written in C++, so any netcode used would need to be written in C or C++ to be useable. No netcodes written for game engines or written in other programming languages were considered.

Using these criteria, three netcodes were chosen to be tested: Yojimbo [15], GameNetworkingSockets (GNS) [16] and RakNet [17]. Important to note is that the original repository for RakNet had received its last update in 2014 and has been archived since 2022. In light of this, a different repository that forked RakNet (which implemented its last changes in 2020) was selected to ensure the project could still be compiled without the source code differing too much from the original RakNet source code.

For the purposes of the experiments performed, each netcode was approached as a black box. As such, as few of the parameters of each netcode as possible were adjusted while ensuring that each netcode operated under similar circumstances (using reliable ordered communication, setting the right IP addresses etc.). As such, all findings from the experiments performed should be taken in context that all netcodes were used out of the box. Additionally, the specifications of the device used for testing can be found in Appendix A.

5.3 Experiment 1

For experiment 1, we seek to answer the following question: Are there significant differences between the performance of different netcodes?

In order to answer this, we need to compare how netcodes perform in terms of accuracy and CPU time in a variety of network conditions. As such, each test performed should apply a unique combination of latency and packet loss. We tested on all possible testing configurations with a latency of 0, 50, 100 or 150 milliseconds, and 0%, 5%, 10% or 15% packet loss. For each set of testing parameters, we performed 41 benchmarks and took the average accuracy and state-netcode execution time respectively. A full set of tables of the performance results for each netcode with the standard deviations provided can be found in Appendix B.

The tables in said Appendix show the raw results of the data used for all experiments described in this section (aside from experiment 3, the data of which is represented with boxplots instead). Notable is that the standard deviations are substantial. This is a natural feature for both the CPU time and the accuracy. The CPU time recorded is the CPU time of an online run of the game subtracted by a local run of the game, thus isolating the CPU time spent on networking. However, as both the online and local run of the game are highly variable, taking the difference of them will result in this variance amplifying. Meanwhile, accuracy experiences a high standard deviation because each netcode has packet loss patterns which will cause it to fail, and increasing the packet loss will increase the likelihood of this pattern occurring. Though, when this pattern occurs first is still variable in each test run.

Figure 7 and 8 show the results of the experiment represented with a heatmap. These heatmaps have been normalized to better show the small differences between results that are clustered on the lower end of the scale[9]. Immediately noticeable is that a lot of the values in Figure 8 are negative. This is a result of how the CPU times are reported. As explained in Section 4.2.2, the CPU time is presented as the difference between the CPU time of the benchmark performed with networking and the benchmark performed without networking. Logically, it should not be possible for a test to be performed in less time with networking then without unless such a test ended early. This is the case for nearly every test with networking performed that is shown to have a negative CPU time value. Confirming this, Figure 7 shows that whenever a negative CPU time is recorded the accuracy recorded is consistently under 10%, and in most cases close to 0. These values cannot be achieved by a test that does not end early. In the worst case scenario where a test instance performs a full test and every prediction fails, a game instance only shows a correct frame once every 7 frames. With the match used for testing taking 5940 frames (99 seconds times 60 FPS), at least $5940 * (1/7) \approx 849$ frames have to be correct. That is a minimum of $849/5940 * 100 \approx 14.3\%$ accuracy for a test that does not end early, a percentage most tests do not reach.

When comparing the heatmaps for each netcode, it can be seen that both in accuracy and CPU time GNS and RakNet differ substantially from Yojimbo. This is mostly because tests using Yojimbo seem to end early far more often when testing with packet loss on 0ms latency then for GNS and RakNet, suggesting that it is not as resistant to packet loss as the other netcodes tested. It can also be seen that Yojimbo is consistently the fastest netcode in terms of CPU time when compared to GNS and RakNet, even in circumstances where its tests did not end early. GSN also seems to have lower CPU time then RakNet on average. Outside of that, the only obvious differences in performance between the netcodes is that GNS performs notably better in terms of accuracy then the rest with 150ms of latency and 0% packet loss and with 50ms latency and 5% packet loss.

[9]The accuracy color normalization was performed using the matplotlib.colors PowerNorm function, with a gamma value of 0.5. The CPU time color normalization was performed using the matplotlib.colors TwoSlopeNorm function, with a vcenter value of 0.

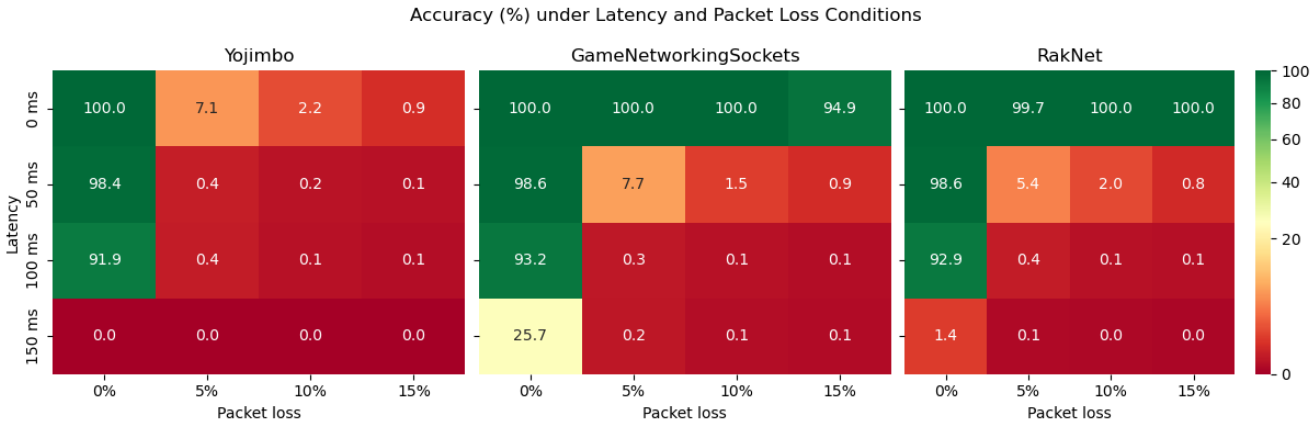

Figure 7: A normalized heatmap showing the accuracy of Yojimbo, GameNetworkingSockets and Raknet under various network conditions.

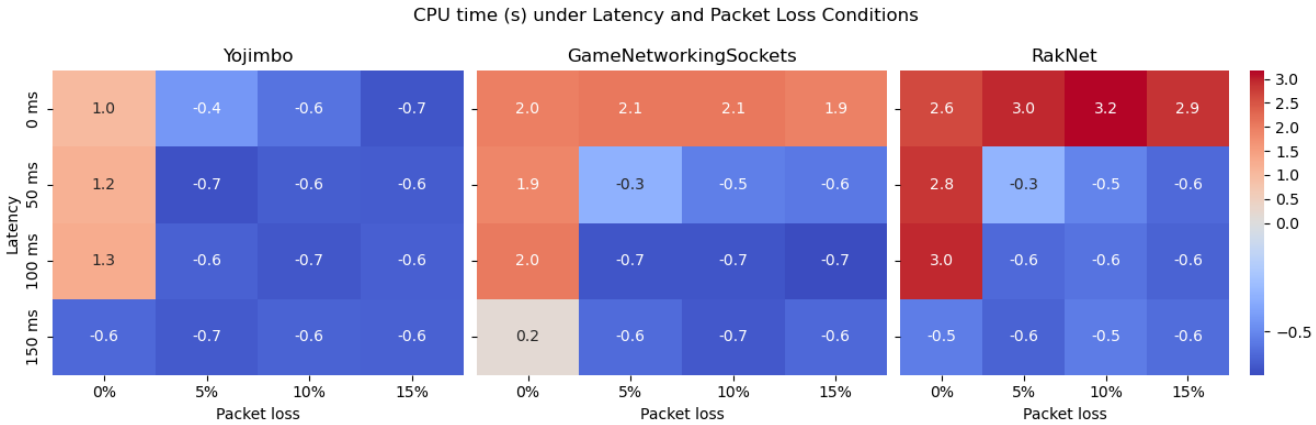

Figure 8: A normalized heatmap showing the CPU time of Yojimbo, GameNetworkingSockets and Raknet under various network conditions.

5.4 Experiment 2

For experiment 2, we seek to answer the following question: Is there a correlation between accuracy and CPU time in the tested netcodes?

In order to answer this, we need to observe the relationship between the accuracy and CPU time of the tested netcodes in a variety of network conditions. For this, we can reuse the results of the tests performed in experiment 1 but compare the accuracy and CPU time of each netcode against each other instead.

As can be seen in Figures 7 and 8, initially there seems to be a strong positive relation between accuracy and CPU time. Whenever the accuracy of a netcode under a certain network condition reaches over 1%, the accompanying CPU time is a lot higher then it is otherwise. This positive relation can not be taken for granted however, as it is largely formed by the occurrence of early stops. When such an early stop occurs, both the accuracy and CPU time of the test performed are logically low due to the test not generating the full amount of frames. If we instead look at the tests performed with an accuracy higher then the 14.3% (as calculated in Experiment 1), we see a different relationship. For both Yojimbo and RakNet, the amount of CPU time seems to increase the lower the accuracy is in those instances. This relation does not seem to hold for GNS however. Our experiments do not provide much data to analyze this relationship for accuracy results over 15% though, so it is hard to say what the exact relationship between accuracy and CPU time is from these results alone.

5.5 Experiment 3

Lastly, for experiment 3 we seek to answer the following question: How robust is each netcode to prolonged bad network conditions?

During experimentation it was noticed that under various network circumstances, netcodes are unable to complete a test set (as discussed in Experiment 1). This leads us with the question if each netcode tested has a different robustness to these network conditions, or if they are all equally non-robust to latency and packet loss.

In order to establish if there is a difference in robustness under the stated conditions in our chosen netcodes, we performed 100 tests for each netcode with a latency of 50ms and 5% packet loss. These network conditions were chosen because in previous experiments, these values showed the greatest variety between netcodes without any test lasting a full match. For each test, we recorded the amount of frames generated before the netcode stopped functioning due to falling too far behind to be recoverable as described in Section 4.1.4.

Figure 9 shows the results of the experiment represented in a Kaplan-Meier plot. As can clearly be seen, Yojimbo is far more likely to fail early on then GNS and RakNet. GNS and RakNet are much harder to distinguish.

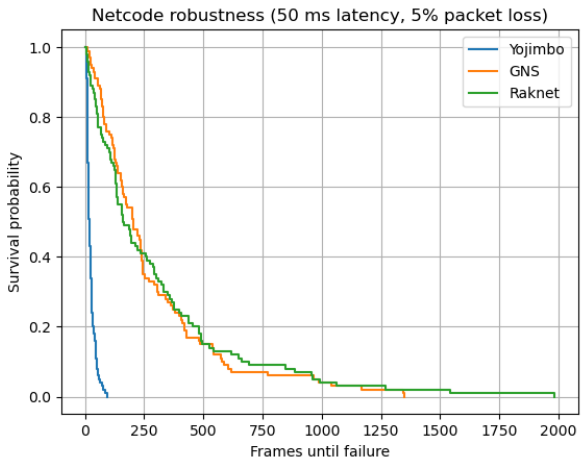

Figure 9: A Kaplan-Meier plot showing the survival probability of Yojimbo, GNS and RakNet failing under 5% packet loss and 50ms latency.

5.6 General Findings

From the results we find a significant difference in robustness between Yojimbo and the other netcodes tested, as well as a significant difference in accuracy and CPU time. As Yojimbo seems to have an overal lower CPU time but also lower accuracy and robustness, this indicates that its implementation may trade in robustness in favour of low CPU times. The documentation provided in each of their repositories backs this assertion up, as Yojimbo offers developers the opportunity to fine-tune many of its parameters by hand whereas GNS and RakNet do so automatically [15] [16] [17]. Additionally, when comparing GNS and RakNet across experiments, its notable that they are only considerably different in respects to their CPU time. This could be a result of RakNet being over a decade older then GNS and as such being less optimized.

It can also be noted that the performance of netcodes in regards to their accuracy and robustness degraded quicker under networking conditions than what we had anticipated. This seems to indicate that either the testing conditions were more straining then was initially expected, or that the black-box approach to fine-tuning the implemented netcodes had a larger effect on their performance then initially assumed.

All in all, this leaves us to conclude that Yojimbo differs significantly in accuracy from the other netcodes tested and that Yojimbo, GNS and RakNet differ significantly in their CPU time under similar network conditions. While we have no conclusive results regarding the relationship between accuracy and CPU time for the netcodes tested, we have shown that Yojimbo is significantly less robust in a certain network condition then GNS and RakNet.

6 Conclusions

In this thesis we presented ButtonCheck, a benchmark to evaluate the accuracy of netcodes under different network conditions.

Using the benchmark, three netcodes were evaluated: Yojimbo, GameNetworkingSockets and RakNet. Our results indicate that there are significant differences in accuracy between Yojimbo and the other netcodes tested. Yojimbo was shown to have the lowest CPU time under the conditions tested, followed by GNS and lastly RakNet. We did not find any convincing evidence to indicate what type of relation exists between each netcode's accuracy and CPU time, but we did show that Yojimbo is significantly less robust under certain network conditions than GNS and RakNet.

Our study shows that different netcodes can produce unique performance patterns in relation to accuracy when tested under different conditions, and that these differences are likely related to their unique implementation strategies. It was also noted that the fine-tuning of netcodes may be more important to a netcode's performance then was initially assumed.

All in all, our research expands the amount of benchmarking options available for developers of competitive multiplayer games. By providing more tools for developers and researchers to compare different netcode implementations, our research aims to encourage a deeper understanding of the various netcodes available.

6.1 Limitations

There are a few limitations concerning this thesis. Firstly, all experiments were performed with an average packet loss burst length of 1 using the Gilbert-Elliot model implementation in TC Netem, meaning that given the implementation at maximum 1 packet was dropped consecutively. The choice of what can be considered an average burst length differs considerably depending on connection conditions and network technologies taken into account. Since there is no singular burst length that accurately captures all scenarios, choosing a larger value would have introduced assumptions about the network scenario. Additionally, since higher burst lengths can create more complex burst patterns, we decided that to minimize moddelling assumptions, it would be best to test each net-statecode with the simplest burst patterns across all network conditions. Given this, the results represent the behaviour of the net-statecodes under isolated packet loss and should not be generalized to environments with higher rates of consecutive packet loss.

Lastly, ButtonCheck has some limitations in its design that cause it to not be useable under all testing circumstances. Mainly, its reliance on TC Netem for network simulation means that it cannot be used on Windows in its current state. While many open source netcodes can be compiled in Linux, some are Windows exclusive and cannot be tested using ButtonCheck. Furthermore, ButtonCheck also does not support peer-to-peer communication.

6.2 Further Research

There are various avenues for further research that this study provides, both in relation to the contributions provided and limitations encountered during experimentation.

It could be interesting to expand Benchy-Fighters to incorporate a more complete character moveset to better capture the full complexity of modern fighting games. It could also be interesting to add multiple characters to include more variety in movesets. Benchy-Fighters currently only encapsulates the bare basics of a fighting game, but it may be interesting to see how various performance metrics scale in relation to the complexity of a fighting game.

Furthermore, adding support for the Unreliable Unordered messaging structure to Benchy-Fighters would allow future research to test potential differences in efficiency in context of a fighting game between Reliable Ordered and Unreliable Unordered messaging regarding CPU time, accuracy and robustness. Adding support to research the differences between performance metrics of a client-server netcode implementation versus a peer-to-peer netcode implementation may also provide valuable insights.

Additionally, performing further testing with less aggressive networking conditions may reveal a clearer relation between accuracy and CPU time. Lastly, studying the difference between more optimized implementations of all the netcodes tested and Benchy-Fighters as a whole could reveal the significance of tuning certain variables in optimizing fighting game accuracy.

A Device Specifications

All experiments were performed in an Oracle VirtualBox virtual machine. The device used to run this virtual machine has the following relevant specifications:

CPU: Intel Core i7 9700

- **Core count:** 8
- **Thread count:** 8
- **Clock speed:** 3000 MHz
- **Cache sizes:**
  - **L1:** 8 x 32 KB
  - **L2:** 8 x 256 KB
  - **L3:** 12288 KB

RAM: 2x Corsair Vengeance LPX

- **Type:** DDR4
- **Size:** 8192 MB
- **Max bandwith:** 1066 MHz
- **SPD ext.:** XMP

Motherboard: Gigabyte Z390 UD

Memory: Crucial P3 1TB

- **SATA type:** SATA-III
- **Speed:** 7200 RPM

The virtual machine has the following specifications:

Operating system: Ubuntu 25.04

Processors: 4

B Other Data

Yojimbo CPU time (s)				
Latency	Packet loss			
	0%	5%	10%	15%
0 ms	1.01 (0.48)	-0.44 (0.23)	-0.57 (0.15)	-0.67 (0.21)
50 ms	1.18 (0.51)	-0.68 (0.20)	-0.64 (0.13)	-0.65 (0.15)
100 ms	1.28 (0.43)	-0.63 (0.17)	-0.66 (0.23)	-0.63 (0.22)
150 ms	-0.61 (0.13)	-0.65 (0.16)	-0.63 (0.16)	-0.63 (0.17)

Table 1: A table showing the raw data and standard deviations from Yojimbo when tested on CPU time.

Yojimbo Accuracy (%)				
Latency	Packet loss			
	0%	5%	10%	15%
0 ms	100.00 (0.00)	7.05 (5.75)	2.20 (2.33)	0.93 (0.89)
50 ms	98.37 (0.43)	0.44 (0.40)	0.18 (0.12)	0.13 (0.10)
100 ms	91.95 (0.59)	0.35 (0.43)	0.14 (0.14)	0.09 (0.08)
150 ms	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)

Table 2: A table showing the raw data and standard deviations from Yojimbo when tested on accuracy.

GNS CPU time (s)

Latency	Packet loss			
	0%	5%	10%	15%
0 ms	1.98 (0.38)	2.08 (0.42)	2.07 (0.45)	1.95 (0.75)
50 ms	1.86 (0.33)	-0.34 (0.30)	-0.53 (0.14)	-0.56 (0.15)
100 ms	2.04 (0.42)	-0.67 (0.26)	-0.67 (0.20)	-0.70 (0.24)
150 ms	0.15 (0.98)	-0.60 (0.15)	-0.66 (0.20)	-0.61 (0.16)

Table 3: A table showing the raw data and standard deviations from GNS when tested on CPU time.

GNS Accuracy (%)

Latency	Packet loss			
	0%	5%	10%	15%
0 ms	100.00 (0.00)	99.99 (0.02)	99.96 (0.04)	94.89 (16.75)
50 ms	98.63 (0.27)	7.70 (9.89)	1.55 (1.67)	0.87 (0.83)
100 ms	93.19 (0.03)	0.29 (0.20)	0.13 (0.12)	0.09 (0.09)
150 ms	25.71 (32.75)	0.24 (0.25)	0.10 (0.08)	0.07 (0.07)

Table 4: A table showing the raw data and standard deviations from GNS when tested on accuracy.

RakNet CPU time (s)

Latency	Packet loss			
	0%	5%	10%	15%
0 ms	2.65 (0.48)	2.96 (0.85)	3.17 (1.24)	2.86 (0.63)
50 ms	2.84 (0.54)	-0.33 (0.38)	-0.51 (0.19)	-0.60 (0.19)
100 ms	2.96 (0.59)	-0.60 (0.15)	-0.57 (0.20)	-0.63 (0.18)
150 ms	-0.54 (0.37)	-0.62 (0.18)	-0.55 (0.16)	-0.59 (0.22)

Table 5: A table showing the raw data and standard deviations from RakNet when tested on CPU time.

RakNet Accuracy (%)

Latency	Packet loss			
	0%	5%	10%	15%
0 ms	100.00 (0.00)	99.68 (2.02)	99.99 (0.02)	99.97 (0.03)
50 ms	98.63 (0.42)	5.39 (5.63)	2.00 (1.91)	0.79 (0.72)
100 ms	92.94 (0.63)	0.36 (0.37)	0.12 (0.13)	0.11 (0.10)
150 ms	1.45 (7.32)	0.05 (0.16)	0.03 (0.08)	0.02 (0.06)

Table 6: A table showing the raw data and standard deviations from RakNet when tested on accuracy.

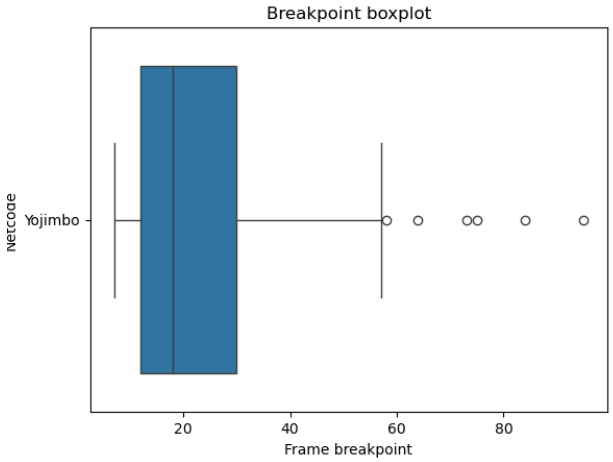

Figure 10: Caption

Figure 11: A boxplot showing the distribution of the test data from Yojimbo when tested on robustness

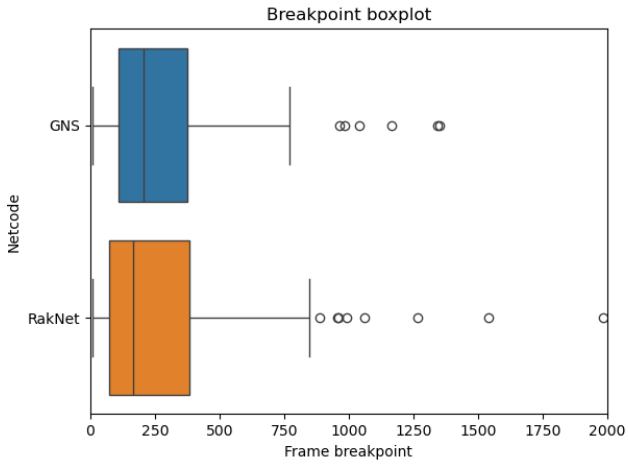

Figure 12: Caption

Figure 13: A boxplot showing the distribution of the test data from GNS and RakNet when tested on robustness

Yojimbo rejection rates

Latency	Packet loss			
	0%	5%	10%	15%
0 ms	6.82% (44)	0% (41)	0% (41)	0% (41)
50 ms	2.38% (42)	0% (41)	0% (41)	0% (41)
100 ms	0% (41)	0% (41)	0% (41)	0% (41)
150 ms	0% (41)	0% (41)	0% (41)	0% (41)

Table 7: A table showing the rejection rates and full amount of tests performed on Yojimbo for experiments 1 and 2 (baseline is 41 tests).

GNS CPU time rejection rates

Latency	Packet loss			
	0%	5%	10%	15%
0 ms	6.82% (44)	0% (41)	2.38% (42)	0% (41)
50 ms	0% (41)	0% (41)	0% (41)	0% (41)
100 ms	0% (41)	0% (41)	0% (41)	0% (41)
150 ms	0% (41)	0% (41)	0% (41)	0% (41)

Table 8: A table showing the rejection rates and full amount of tests performed on GNS for experiments 1 and 2 (baseline is 41 tests).

RakNet CPU time rejection rates

Latency	Packet loss			
	0%	5%	10%	15%
0 ms	0% (41)	0% (41)	0% (41)	4.65% (43)
50 ms	2.38% (42)	0% (41)	0% (41)	0% (41)
100 ms	0% (41)	0% (41)	0% (41)	0% (41)
150 ms	0% (41)	0% (41)	0% (41)	8.89% (45)

Table 9: A table showing the rejection rates and full amount of tests performed on RakNet for experiments 1 and 2 (baseline is 41 tests).

Robustness rejection rates

Yojimbo	1.96% (102)
GNS	2.91% (103)
RakNet	12.28% (114)

Table 10: A table showing the rejection rates and full amount of tests performed on each netcode for experiment 3 (baseline is 100 tests).

C Program Installation

The repositories for ButtonCheck and Benchy-Fighters contain information on the building process and point to further resources for building their sublibraries and additional programs provided to use for the benchmarking setup. Instead, here we will focus on the installation process of Boost version 1.88+. As of writing, all sublibraries required to run this benchmark are supported by the latest Ubuntu LTS except for Boost. This project uses Boost V2 which was properly introduced in Boost 1.88, though the current LTS only goes up to Boost 1.87. In case you work on a Linux installation that does already support Boost 1.88+, this can be safely ignored.

In order to install Boost to use for this process, first go to the Boost website and install any Boost version equal to or greater then 1.88. Place the installed file wherever you prefer. Then, unpack it and run the following call inside the top layer folder:

```
./bootstrap.sh
```

After that command finishes, run the following call from the same folder. Note, this call only prepares the Boost libraries required for this project:

```
./b2 --with-filesystem --with-process link=shared \\
runtime-link=shared variant=release install
```

Lastly, run the following call to update your shared libraries to include the installed Boost version (this will likely require admin privileges):

```
ldconfig
```

Below is an example of a network emulation file as included in ButtonChecks repository:

```
[
        {
                "//comment": "use this structure for fully random packet loss",
                "Frame": "0",
                "Delay": "0ms",
                "PacketLoss": "5%"
        }
]
[
        {
                "//comment": "use this structure when applying
                              the Gilbert-Elliott packet loss model",
                "Frame": "0",
                "Delay": "0ms",
                "AvgBurstLength": "5",
                "ErrorRate": "5%"
        }
]
[
        {
                "//comment": "alternatively, use this structure to define the chance to enter
                              the bad state and to stay in the bad state by hand. The loss in both
                              the good and bad state will have to be defined as well",
                "Frame": "0",
                "Delay": "0ms",
                "EnterBad": "20%",
                "ExitBad": "50",
                "GoodLoss": "2%",
                "BadLoss": "90%"
        }
]
```

References

[1] SteamDB, "Steam charts," accessed: Aug. 03, 2026. [Online]. Available: https://steamdb.info/charts/?sort=peak

[2] R. Rogers, *Understanding Esports: An Introduction to the Global Phenomenon*. Lexington Books, 2019, ch. Chapter Nine, Fighting Games and Social Play, Lee K. Farquhar.

[3] enpicie, "Findmyfgc," accessed: Aug. 12, 2026. [Online]. Available: https://www.findmyfgc.cc

[4] R. Pusch, "Explaining how fighting games use delay-based and rollback netcode." [Online]. Available: https://arstechnica.com/gaming/2019/10/explaining-how-fighting-games-use-delay-based-and-rollback-netcode/

[5] "Multiplayer networking resources, a curated list of multiplayer game network programming (netcode) resources," accessed: Aug. 12, 2026. [Online]. Available: https://multiplayernetworking.com

[6] P. Mieschke, "Deterministic lockstep in networked games," Master Thesis, Hochschule der Medien, 2024.

[7] M. Ahmed, S. Reno, M. R. Rahman, and S. H. Rifat, "Analysis of netcode, latency, and packet-loss in online multiplayer games," in *2022 International Conference on Augmented Intelligence and Sustainable Systems (ICAISS)*, 2022, pp. 1198–1202.

[8] M. Huynh and F. Valarino, "An analysis of continuous consistency models in real time peer-to-peer fighting games.(2019)," 2019.

[9] A. Ehlert, "Improving input prediction in online fighting games," 2021.

[10] J. Eickhoff, "Polka: A differentiated deployment system for online and streamed games, meta-verses, and modifiable virtual environments," Master's thesis, TU Delft, 2024.

[11] E. Stroiu, "Net-celerity: A benchmark for player activity analysis of gaming network libraries," Ph.D. dissertation, Bachelor's Thesis. Vrije Universiteit Amsterdam, 2024.

[12] R. L. Chandra, P. Querl, D. Berkaoui, K. Castermans, and H. Nacken, "Comparison of open-source networking libraries for unity engine in higher education," *Journal of Computer Science and Technology Studies*, vol. 6, no. 3, pp. 65–75, 2024.

[13] R. Pusch, "Explaining how fighting games use delay-based and rollback netcode," 2019.

[14] Arc System Works, "Guilty gear -strive-," Video game, 2021.

[15] G. Fiedler, "Yojimbo," 2025, accessed: Aug. 12, 2026. [Online]. Available: https://github.com/mas-bandwidth/yojimbo

[16] Steam, "Gamenetworkingsockets," 2025, accessed: Aug. 12, 2026. [Online]. Available: https://github.com/ValveSoftware/GameNetworkingSockets

[17] O. VR, "Raknet," 2025, accessed: Aug. 12, 2026. [Online]. Available: https://github.com/miniclip/RakNet