An analysis of continuous consistency models in real time peer-to-peer fighting games

Martin Huynh, Fernando Valarino
Authors
Martin Huynh, Fernando Valarino

Title
An analysis of continuous consistency models in real time peer-to-peer fighting games.

Supervisor
Kamilla Klonowska

Examiner
Eric Chen

Abstract
This study analyses different methods of maintaining a consistent state between two peers in a real time fighting game played over a network. Current methods of state management are explored in a comprehensive literature review, which establishes a baseline knowledge and theoretical comparison of use cases for the two most common models: delay and rollback. These results were then further explored by a practical case study where a test fighting game was created in Unity3D that implemented both delay and rollback networking. Networking strategies were tested by a group of ten users under different simulated network conditions and their experiences were documented using a Likert-style questionnaire for each stage of testing. Based on user feedback it was found that the implemented rollback strategy provided an overall better user experience. Rollback was found to be more responsive and stable than the delay implementation as network latency was increased, suggesting that rollback is also more fault tolerant than delay.

Keywords
Peer-to-peer, state management, continuous consistency, games, online, networking, Unity3D
# Contents

1 Introduction
   1.1 Background 4
   1.2 Aim and Purpose 4
      1.2.1 Relevance to video games 5
      1.2.2 Other Implementations 5
   1.3 Research Questions 6
   1.4 Limitations 6

2 Methodology
   2.1 Literature Review 8
      2.1.1 Scientific and Scholarly Articles 8
      2.1.2 Textbooks 8
      2.1.3 Supplementary Sources 8
      2.1.4 Literature Search Results and Selection Process 8
   2.2 Case Study 10
      2.2.1 Development of a test game 10
      2.2.2 Procedure 11
      2.2.3 Questionnaire 12
         Ethical Considerations for the Questionnaire 12

3 Literature Review
   3.1 Overview of Common Network Architectures 13
      3.1.1 Client-Server Architecture 13
      3.1.2 Peer-to-Peer 13
   3.2 Delay-based Networking 14
      3.2.1 Lockstep Protocol 14
      3.2.2 Asynchronous and Pipelined Lockstep Protocols 16
   3.3 Rollback-based Networking 17
      3.3.1 Rollback in Databases 17
      3.3.2 Rollback Networking in Video Games 17
      3.3.3. Prediction and GGPO 19
   3.4 Summary 21

4 Case Study
   4.1 Background 22
   4.2 Gameplay mechanics of a test fighting game 22
   4.3 Implementation in Unity3D 24
      4.3.1 Game simulation and State handling 24
      4.3.2 Networking 25

5 Results
   5.1 Literature Review 27
   5.2 Case Study 28
1 Introduction

1.1 Background

Netcode is a term often used haphazardly within the video game community to describe the overall implementation of networking in a video game. The term can be commonly seen or heard when players comment on or discuss synchronisation issues occurring whilst playing an online video game. Players will often use sweeping statements about “bad netcode” in a game, when the causes of such issues could be several things completely out of the game’s control. Such causes can include (but are not limited to): high latency to the server, packet loss, network congestion for either the client or server, or even external factors completely unrelated to network quality, for example frame render time and inconsistent frame rates. Most often it is a combination of these factors that cause network instability.

Networking/synchronisation issues between clients are sometimes eased through methods of lag compensation. These are networking techniques implemented to disguise network stresses and aid client performance. For games that require fast, twitch responses such as fighting games, competitive shooters and racing games, optimised networking performance is of chief importance since these types of games rely on their multiplayer aspect for longevity. If the networking performance of a game is poor on release, there can be significant backlash from consumers [1].

Some method of lag compensation will be present in all online video games, regardless of network model. Team-based online games most often use a client-server networking topology, where clients will connect to a dedicated server, which acts as an authority and relay for client updates (see Figure 1).

However, for most one-vs-one games, such as fighting games, peer-to-peer (see Figure 2) is often the preferred networking topology. Arguably one of the biggest differences between the two topologies is that peer-to-peer implementations will lack an authoritative entity to maintain consistency in state between each client.

This therefore introduces the following problem: as there will be inherent inconsistencies in any network, two clients will always be somewhat ‘out of sync’. Ideally, two clients should have identical, or as close to identical states as possible and provide essentially a real time response. This problem is also present in client-server implementations but is amplified in peer-to-peer due to the lack of authority.
In a peer-to-peer network, if two clients are desynchronized, there is no authoritative entity to dictate the correct state, and neither clients should have authority over the another. There is no hierarchy and there is now a problem of consensus between clients. This is where a suitable continuous consistency model is required to maintain state between two clients.

1.2 Aim and Purpose

The aim of this work is to document, discuss and compare different methods of maintaining a consistent state between clients in a real-time peer-to-peer networked application where there is no authoritative entity. This is done in the form of a literature review documenting current solutions, as well as a case study comparing different implementations with regards to user experience.

1.2.1 Relevance to video games

Little research into concurrency and state management control can be found in relation to real-time non-authoritative models, such as video games. There is especially little research comparing different peer-to-peer state management protocols in terms of user perception. This is hugely important in the context of real-time applications as different solutions will have different effects on the user experience. A comparison of solutions, when user experience is considered, could reveal what is potentially best for users and not necessarily the “correct” implementation since video games are high interaction, high tempo, and require a degree of “fairness” towards all players.

1.2.2 Other Implementations

Despite this work mainly focusing on real-time implementations commonly found in video games, the problem of state management is also relevant to other applications. Most notably, database management also requires strict protocol with regards to concurrency control, since unhandled, concurrent database transactions can cause a slew of problems [2]. One of the most promising solutions for real-time peer-to-peer state management is a derivative of a well-established technique used for databases (rollback) [3].

Blockchain is another relatively new technology that is both peer-to-peer and by the nature of its design, provides very strict concurrency control. In the context of cryptocurrencies, the speed of block additions (known as transactions) to the blockchain can vary wildly depending on the cryptocurrency being used [4]. This is due to their requirement for mining nodes to verify the transactions taking place. Incoming transactions are generally prioritised by miners based on how much of a fee the miner will receive by carrying out the verification task.
1.3 Research Questions
The following research questions will be discussed throughout this paper. The predominant question throughout this paper will be:

“How can a consistent state be maintained between two clients over a network, where no client has authority over another?”

To answer this question, a baseline of theory is established, and current research and real-world continuous consistency models are discussed and compared within a literature review. This question then leads into a follow up question:

“How do different continuous consistency models compare to one another in terms of performance and failure tolerance with regards to user perception?”

This second question was explored in a case study where the two most used continuous consistency models were implemented and compared within a test fighting video game. The case study was aided by a short background/literature review chapter that explores previously established results of other studies.

As both authors have abundant experience with video games, and especially fighting games, the following predictions were made to establish the expected results for the case study:

1. Player inaccuracy would increase with latency and packet drop.
2. Round length will increase with latency and packet drop.
3. Delay networking will cause more frustration than rollback networking.
4. Rounds with lower latency will be felt as more responsive.

1.4 Limitations
The case study only compared the two most used continuous consistency models within fighting games. Specifically, delay-based networking (or lockstep), was compared to rollback-based networking. Of course, user experience is always subjective, therefore it was important for the test group providing feedback to be composed of a variety of different potential users, i.e. those who are experienced with video games ranging to those who are not.

The implemented models in the case study included options to inject simulated network issues to show how each model reacted to these inconsistencies. The two networking issues that were tested were network latency and packet loss. Although inconsistent frame render times and frame rates are often closely compared to network issues, they were not included in the case study as they are caused by poor quality hardware or problems with the system itself as opposed to network problems.

The case study focused mainly on the user perception of network stresses for delay-based and rollback-based networking strategies. While log data was collected for each stage, for example the input and state data of each frame of a round, this data proved not to be useful in the analysis. Log data is sparse even under similar testing conditions, as test subjects approach the game using different strategies, and thus provides limited insight on the perception and performance of the consistency models. Log data proved useful only to guarantee that consistency had indeed been achieved.
The study does not propose a new way of implementing these models, thus, additional optimizations that would be possible in a game designed with only one of these models in mind, such as only employing rollbacks during times the player is not able to make another input, are not accounted for.
2 Methodology

2.1 Literature Review

As interactive media and video games tend to be a multidisciplinary effort, a broad array of scientific works regarding networking, interaction design, concurrency and even database transactions helped contribute to an understanding of the suitability of consistency control models for every scenario in which they are used. Therefore, the first stage in answering the proposed research question involves a comprehensive study into existing literature regarding these different models. The overall aim of the literature review was to establish a baseline knowledge which could be used as a reference for the case study implementation. The literature review covers the theory of current solutions, as well as examines the relationships between seemingly unrelated implementations (à la rollback networking in video games and databases).

The literature review uses sources from three different categories.

2.1.1 Scientific and Scholarly Articles

These include scientific articles found on a multitude of digital libraries including the Association for Computing Machinery (ACM) digital library and Google Scholar. Although not necessarily a library, ResearchGate, a popular social media platform for academics, was also useful for obtaining scientific articles. Academics can directly publish their papers on ResearchGate, and it contains useful statistical information such as number of citations by other articles, reference lists and more. Scientific articles provide a valuable outlook into what research has been, and is currently being conducted in the topic area.

2.1.2 Textbooks

Textbooks can provide an insight to already well-established implementations, as opposed to the often experimental and theoretical solutions offered by scientific articles. Textbooks are accessible through public libraries, as well as through the library at Högskolan Kristianstad.

2.1.3 Supplementary Sources

These sources present a more general outlook on the topic of this work. They include developer blogs, news articles, tutorials, conference talks and panels, and interviews. These sources are not peer-reviewed like scientific articles, but they provide real-world insight into the effects of consistency control models in modern applications, especially in video games. Likewise, developer blogs from well-established engineers and developers are a haven of knowledge that should absolutely be acknowledged.

2.1.4 Literature Search Results and Selection Process

Sources for the literature review were found by breaking down the first research question into keywords and then expanding them by using their synonyms. These keywords were then used in different combinations as search terms during the process of finding relevant articles and documentation (see Figure 3). The main source of literature was the ACM digital library, as well as Google and ResearchGate. Search terms were often used with AND and OR expressions to expand or focus results depending on the combination. Generally, the OR expression resulted in the most articles found, followed by the AND expression, and finally using “phrase” searches.
resulted in the least number of articles. Although phrase searches tended to result in more relevant results to the study.

The ACM library also allows for advanced searches, which were, for example, used to limit a search only to article headings, abstract or full text. This proved to be useful in that some searches would result in thousands of results when full text was used as a parameter, only to be narrowed down to a few hundred with the “abstract only” limit, and then to single digits when focused on the title alone. Generally, the use of logical operators provided better focus on relevant articles, but using these limiters, especially the “abstract only” limiter, also provided a somewhat effective means of narrowing down search results for articles.

![Figure 3: Keyword processing method.](image)

During the literature search, a significant number of articles were found with relation to networking and state management within video games. However, most of these articles were not relevant to the research question. There appeared to be little in the way of research into peer-to-peer state management for video games, with research being disproportionately focussed on the more popular client-server model. This could simply be because peer-to-peer architectures are not used as much as client-server architectures for video game implementations.

Articles were filtered using a simple flowchart (see Figure 4) to determine if the contents were relevant. First the titles were assessed, followed by the abstract, and finally the full text. If at any point the article was not relevant to the research question, the article was discarded. The ACM digital library, Google Scholar and ResearchGate also provides statistics for each paper, such as the number of citations from other articles, which proved useful in finding credible articles. Likewise, when one article was deemed to be relevant, further articles could be found within their own references. In the same way, Wikipedia also proved to be useful in the search. Although content from Wikipedia articles was not used, their references could point towards other scientific articles that may not have been available through ACM.
2.2 Case Study

2.2.1 Development of a test game

To establish a platform for comparing different consistency control models, a game was developed using the Unity3D game engine in the C# programming language, taking into account certain conditions:

1) The game must be playable by more than one person, so that it may be made playable over a network.

2) The game must be fast paced, i.e. inputs must be time sensitive, so that delays caused by network effects are perceptible to users. This is to ensure that efficiency in the networking strategy is consistently achieved.

3) The game state must be deterministic, wherein, transforming the game state through an action or input must reliably result in the same state being reached. Because of this, the game state becomes reversible; if all player actions and inputs in the game are recorded, the same simulation state can be achieved.

Taking this into account, it can be argued that a fighting game fulfils all these conditions as two players compete in simulated combat through a limited set of moves and actions that may be executed sequentially if performed with the right timing. Since correctness and timing are crucial for the fighting game experience, aberrations in the game experience caused by network effects should strongly impact said experience.

Therefore, a fighting game was developed. The game consists of two identical characters that battle one other to deplete the opponent character’s health using combat moves, all under a limited amount of time. Whoever depletes the opponent’s health bar first, wins. If neither health
bar is depleted and time has elapsed, the player with the most health wins. If both players have the exact same amount of health or deplete each other’s health bars at the exact same time, the game results in a tie.

The game was made to be played over a network connection, with two networking models implemented, one which employed a delay-based strategy, and another which employed a rollback strategy. The game was made to allow for injected artificial latency, as well as pseudo-random packet drop to be applied so that different test environments could be easily set up.

2.2.2 Procedure
The game was presented to a test group of 10 participants with varying degrees of experience with video games. These participants were made to play the test fighting game over a network with two machines on a wired connection to minimize the real latency between the two machines and allow controlled values to be established. It is important to note that both simulations could not be run on one machine as only one simulation can be controlled at one time, meaning only one player will be able to move. Both players used a PlayStation 4 controller.

Tests for delay and rollback followed the same procedure, participants first played two rounds under “perfect” circumstances, i.e. no added latency or packet loss, as a baseline. Participants then played two rounds per stage, with each stage having a different network stress configuration. In between each stage, the participants were asked to note their experiences with regards to the feel of the game. The different network stress settings are as described below. The tests for delay and rollback were carried out separately, i.e. stages for each were not mixed. Although settings were applied randomly for each network configuration to prevent any obvious patterns from emerging.

1. No artificial delay, 5% packet loss
2. 50ms artificial delay, no packet loss
3. 50ms artificial delay, 5% packet loss.
4. No artificial delay, 10% packet loss.
5. 100ms artificial delay, 0% packet loss.
6. 100ms artificial delay, 10% packet loss.
7. 200ms artificial delay, 0% packet loss.
8. 200ms artificial delay, 10% packet loss.
9. No artificial delay, 20% packet loss.

The latency values for each stage represent good, adequate and bad connections for 50ms, 100ms and 200ms respectively. These values were chosen as they were the milestones used in similar studies, such as those performed by Beigbeder et al., as well as Pantel and Wolf when testing the effects of latency on user perception in the First Person Shooter game Unreal Tournament 2003 [5] and several racing games [6]. Beigbeder et al. also consider packet loss values of 5% to be high but also found that it did not affect the user’s ability to perform, therefore the values of packet loss for this case study begin at 5% and rise to 20%. Packet loss of 20% was only tested with no artificial delay as packet loss this high should theoretically result in very poor
performance regardless of latency. Time consideration was also a part of this decision as participants must already complete 18 stages in total with two rounds played for each stage.

### 2.2.3 Questionnaire

Participants answered questions after each round played during the test, where they qualitatively evaluated criteria about their experience based on a 5-point Likert-type scale. The questions were as follows:

This first question was asked before the tests to aid with candidate selection.

*On a scale from 1-5, how experienced are you with video games? With 1 being not experienced at all and 5 being very experienced.*

Questions 1 through 4 were asked in between each stage of testing with the following possible answers:

- [Strongly disagree]
- [Disagree]
- [Neutral]
- [Agree]
- [Strongly Agree]

**Q1.** I was able to react to my opponents’ actions quickly.  
**Q2.** The game was responsive to my inputs.  
**Q3.** The network connection was unstable.  
**Q4.** I played well this round.

### Ethical Considerations for the Questionnaire

As testing for the case study was carried out by having real users play a test game and answer questions based on their experience, certain ethical issues were considered. Firstly, testers all remained anonymous, and all information provided by testers were used strictly for the purpose of the case study. No personal information was requested from the testers. Likewise, each tester gave their consent before taking part in the testing procedure and answering the questionnaire.

Questions were also designed to be as impartial as possible to prevent bias in answers. The questionnaire was in a Likert-scale style where testers were required to note whether they agree or disagree with a statement about their experience.

It was also necessary to eliminate biases when selecting users. Unfortunately, testers could not be completely randomly chosen, but care was taken to ensure that there was a somewhat equal distribution of testers with varying degrees of experience with video games, and therefore prevent potential skewed data and instead provide a well-rounded sample of data. This was done with establishing the participants experience as the first question of the questionnaire before the tests began.
3 Literature Review

3.1 Overview of Common Network Architectures

3.1.1 Client-Server Architecture

Within multiplayer video games, game state consistency is of utmost importance. In a typical client-server implementation, the state is handled by the server. Gabriel Gambetta, in his series on developing a fast-paced multiplayer shooter describes this implementation as an ‘authoritative server and dumb clients’ where the central server is controlling the whole simulation, and clients are simply spectators of the game with privileges to allow for remote input [7]. Another way of describing the situation is that the game state of a client should not be trusted. Instead the state should always be confirmed by the server, which is constantly processing incoming data from all clients, and therefore can always provide an accurate representation of the game.

Gambetta offers further explanation by using player position as an example:

‘You also don’t trust the player with its position in the world. If you did, a hacked client would tell the server ‘I’m at (10,10)’ and a second later ‘I’m at (20,10)’, possibly going through a wall or moving faster than the other players. Instead, the server knows the player is at (10,10), the client tells the server ‘I want to move one square to the right’, the server updates its internal state with the new player position at (11,10), and then replies to the player ‘You’re at (11, 10)’.” [7]

This example also provides another reason as to why client state should not be trusted: cheating. If a client state is trusted, this opens the possibility of a player manipulating the commands being sent to the server, and thus give themselves an unfair advantage. By changing the way in which clients issue commands (requestive commands instead of assertive commands), this type of cheating can be nearly completely avoided.

In summary, a game using client-server should rely on the server itself to keep track of the game state. Clients’ local states should not be trusted, and commands sent to the server should be vague in that they should not contain any discrete data pertaining to the local state of the player (as seen in Gambetta’s example).

3.1.2 Peer-to-Peer

In a peer-to-peer (P2P) networked system, there is no authoritative entity. Instead, data must be sent to and processed by every other client. This inherently means that state synchronisation is more difficult to implement than in a client-server architecture since, on the surface, there does not seem to be a way to resolve inconsistent states between clients.

In its most basic implementation, each peer is responsible for its own state. Peer A has no bearing on the state of Peer B, and so when Peer A needs to update its state, it must send its new state to all other peers in the system; the same can be said for all other peers. There is an assumption that each state is correct, and so this opens up opportunities for state manipulation and cheating. At any one time, one peer will be seeing multiple states: the state of their own avatar, as well as the states of any other players rendered on their screen. This is due to the latency of other peers when sending updates. The result is desynchronization between clients, which can lead to visible ‘jumps’ or ‘stutters’, otherwise known as lag.
Pellegrino and Dovrolis, in their analysis of bandwidth requirements in different networking topologies, found that the bandwidth required per client in a peer-to-peer topology is double that of a client in a client-server architecture [8]. For a client-server architecture, the bandwidth of the server itself scales quadratically with the number of clients connected, whereas for P2P the bandwidth requirements scale linearly with the number of players since there is no bottleneck of a server [8].

This analysis shows that for games with many players, despite the requirement for a server with high bandwidth, client bandwidth requirements are not affected as much as in P2P. This solidifies the general consensus that team-based online video games are more suited to a client-server model, but for games that do not have a requirement for many players, P2P networks can prove to be just as performant, or even better.

Client-server architectures, compared with P2P, also tend to be simpler solutions to implement since clients will only be required to communicate with a single entity. Although this means that the latency experienced by each client is increased due to the server acting as a relay. A peer-to-peer architecture bypasses this, since each client sends their data directly to other clients. In fact, the potential lower latency could be why P2P topologies are favoured in many one-vs-one games since there is both a lower player count requirement, as well as a need for as real-time response as possible.

### 3.2 Delay-based Networking

#### 3.2.1 Lockstep Protocol

The lockstep protocol is a simple method of state synchronisation for a P2P network. It was designed by Nathaniel Baughman and Brian Levine as a solution to dealing with the lookahead cheating issue present in a basic stop-and-wait networking protocol. In a normal stop-and-wait solution, all clients must submit their inputs, and wait for all incoming inputs before the overall state of the system can be updated (see Figure 5). Lookahead cheating is when a client will wait for all other clients to submit their inputs before executing its own, thereby gaining an unfair advantage.

The lockstep protocol requires all client to announce a cryptographically secure one-way hash of their decisions (defined as a commitment). Once all commitments have been declared, players then will reveal their commitments to one another [9]. Since the decisions are declared through a one-way hash, their legitimacy can be proven by any of the other clients by simply comparing the plaintext decision with the hashed counterpart received beforehand. This prevents any clients from using a lookahead cheating strategy, as only the current turn data is available to each client [9].

Real-time strategy (RTS) games commonly implement lockstep-based networking. Real-time strategy games are games in which a player will have control over an army of characters (units) in a “top down” view. Typically, the player will order a set of characters to carry out some task,
for example attacking an enemy unit, building resources and more. One such game series is *Age of Empires*. Mark Terrano and Paul Bettner, network developers on *Age of Empires 1* and *2*, have documented their network design architecture and implementation. Since *Age of Empires* (and other RTS games) can contain in excess of hundreds of individual ‘units’ (animals, soldiers, buildings, etc...) per player in each match, they determined that it was not optimal to send the state of each unit to other clients via the network. Instead each client runs the exact same simulation, with each process identical commands issued by each client at the same time [10]. This means that the game state for each player should theoretically be identical. Of course this is not such an easy task due to uncontrollable elements like network latency or packet loss.

Terrano and Bettner’s implementation followed a traditional lockstep implementation, using scheduled communication ‘turns’ to issue commands. A turn would last approximately 200 milliseconds, with each client sending commands during this time. At any point in time, commands would be processed for one turn, with new commands being received and stored for the next turn. Processed commands were sent out for execution two turns in the future [10]. Not only did their implementation provide a stable online environment for players, but it also provided added security since the system required all clients to have identical states. “Any simulation that ran differently was tagged as "out of sync" and the game stopped.” [10]

However, due to the nature of the lockstep protocol, the speed of play was limited by the slowest connection since other clients must wait until they have received enough information to carry out a turn. As all clients were required to know the state of all other clients, they were limited by the time taken for other clients to update. This performance penalty was also highlighted in the initial proposal by Baughman et al [9].

This penalty was mitigated by the slower pace of RTS games. Terrano and Bettner explained that, for *Age of Empires*, a latency of 250ms was not even noticed. Only beyond 500ms did latency begin to be noticeable. They also note that players would become accustomed to latency and develop a “mental expectation” of the latency between command executions and the on-screen response [10]. So not only were there extremely flexible latency limits, but once players were settled in, they adapted to the delay in their gameplay.

This result is also further proven by Mark Claypool in his study on the effects of latency in RTS games [11]. After conducting thorough testing on three different highly successful real time strategy games, Claypool found that network latency did not affect users until approaching and exceeding 500ms, a result similar to Teranno and Bettner’s. Claypool even notes that with latencies exceeding 500ms, players could still react accordingly to the delay, thereby confirming the existence of the mental expectation documented by Teranno and Bettner. Players were only significantly hindered by latencies of 1000ms and over [11].

There are also other downsides to the lockstep protocol, and specifically with Terrano and Bettner’s implementation as explained by Glenn Fiedler, a veteran network developer in the games industry. He notes the difficulty of ensuring that each turn plays out identically for each client where a situation could arise where one client’s units will move slightly differently or take a different path to the final location of the turn, and perhaps arrive sooner or later than the other client [12]. This is not dissimilar to the butterfly effect, where one small change can eventually cause huge differences in the future.

Lockstep is also used in faster paced games. In his talk at the Game Developers Conference, Michael Stallone from NetherRealm Studios discusses the studio’s basic implementation of
lockstep as their method of state synchronisation in both of their major fighting game franchises *Mortal Kombat* and *Injustice*. In their implementation, only player inputs are sent over the network. Once a player performs an input, there is a delay as the input is sent to the opposing player. Once that input has been received, both clients render the corresponding move at the same time [13].

Stallone’s implementation of lockstep for their games proved to be inadequate. In fact, his presentation is about why the studio moved away from delay-based networking to *rollback-based* networking. Their implementation used *dynamic input latency*, where there was a variable delay depending on the network latency being experienced the client. Stallone lamented this implementation decision, stating that due to inconsistent and high input latency, the player-base was not happy. For fighting games, responsiveness and consistency are extremely important. The dynamic input latency was causing inconsistencies with player inputs as the network latency fluctuated [13]. Fighting games are inherently fast paced and executing combos can require a significant amount of dexterity and muscle memory. With their lockstep implementation, a combo may have had to be executed with slightly different timing each time depending on the network latency, leading to unpredictability and potential dropped inputs on behalf of the player [13].

### 3.2.2 Asynchronous and Pipelined Lockstep Protocols

The vanilla lockstep protocol is suitable for non-time critical applications, but if used for a real-time application then either turns should be kept short by limiting the amount of data required for a turn to count, or each turn should have a time limit. This could help keep the simulation moving forward and prevent noticeable freezes to the game state, but the system will still be running at the pace of the slowest client.

Baughman and Levine define an optimization called the *Asynchronous Lockstep Protocol*. Asynchronous lockstep “relaxes the requirements of lockstep synchronization by decentralizing the game clock” [9], which allows clients to advance their clock regardless of the states of one another. However, when one client interacts with another they enter into a “lockstep-style” mode. Lockstep interactions are determined by a *sphere of influence* for each client which they define as “the area of the game that can possibly be affected by a player in the next turn — and therefore potentially require resolution with other player decisions” [9]. With each client having its own sphere of influence, the overall system can determine if one client will interact with another. If so, then the clients must wait for one another to commit their turns to continue, otherwise they should proceed asynchronously. This ultimately improves network performance, since clients will be unaffected by another client’s slower connection unless they are directly interacting with them.

A further development of lockstep can be seen in the *Pipelined Lockstep Protocol* proposed by Lee et al [14]. The protocol further optimizes the Asynchronous Lockstep by relaxing the order by which a hashed commitment and plaintext decision must be sent. Lee at al. provide the reasoning that some situations do not require an ordered set of commitments, i.e. one client doesn’t necessarily require the commitment of another to continue (interaction resolution) [14]. This means that if there is no conflict between a client’s current to-be hashed commitment and other clients’ current plaintext decisions, then further commitments should be permitted ahead of their next plaintext decision declaration. Hashed commitments are *pipelined* in a buffer, which are sent onto the network in order of arrival (first-in-first-out).
3.3 Rollback-based Networking

Rollback-based networking is commonly used as an alternative to delay-based networking. It is a more complex to implement, but in optimal conditions it can both effectively simulate a feeling of ‘real-time’ gameplay and provide a much more predictable input latency. The term ‘rollback networking’ is commonly associated with fighting games as it is the networking solution that caters towards their requirement for fast, twitch reactions to onscreen stimuli. It should be noted that, although commonly associated with fighting games (and peer-to-peer implementations) forms of rollback networking are also commonly used in client-server architectures and for many other genres of video games.

3.3.1 Rollback in Databases

Rollback implementations in video games are similar to that in database management systems. In databases, “schedulers” are employed to organise transactions. There are essentially two different types, aggressive and conservative. Conservative schedulers implement delays to transactions in order to carry them out serially, whereas aggressive schedulers try to schedule transactions immediately. Implementations of aggressive schedulers must include rollback functionality, since a situation may arise in which a series of active transactions cannot be executed in a serializable fashion. In these situations, the scheduler can only fix the erroneous state by aborting scheduled transactions and returning to a previously stable state, thereby rolling back the database.

Generally, in databases, when a rollback is performed, there is always a risk of rollbacks to cascade, that is, the abortion of one transaction T1 may require the abortion of other transactions that rely on the result of T1 [2]. Most SQL-based relational databases have some kind of implementation of savepoints, which act as a point to which a transaction can be rolled back [15].

3.3.2 Rollback Networking in Video Games

Much like in database implementations, rollback implementations in video games also must include some form of savepoint, as will be explained below. Unlike database rollbacks, online video games that implement rollback networking will always have one or more clients rolling back consistently. This is simply because of network delay on the state of both clients.

Before rollback networking for video games can be further explained, it is important to understand the concept of frames. Frames are used as a measure of time (where one frame represents 16.6ms at a frame rate of 60 frames-per-second), for example, an input latency of four frames implies a delay of 66.4ms from the moment of input on the controller to the moment said input is displayed on-screen. In fighting games each move has a set of start-up frames that show a character’s wind-up animation leading up to a hit, a set of active frames where a move can affect (hit) the opponent player and recovery frames where the character is vulnerable to the attacks of the opponent player (see Figure 4).
Below is a basic example of how rollback works within a fighting game.

<table>
<thead>
<tr>
<th>Frame 1</th>
<th>Client A inputs a command. Their character begins their start-up animation which takes 8 frames.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame 3</td>
<td>Client B receives a packet containing client A’s input. Client B is 3 frames ‘behind’.</td>
</tr>
<tr>
<td>Frame 4</td>
<td>Client B ‘rolls back’ and simulates the first 3 frames and ‘resumes’ the animation of client A’s character from the current frame. The simulation is performed within the time of one frame.</td>
</tr>
<tr>
<td>Frame 5</td>
<td>Client A and Client B are now synchronised, with 4 frames remaining in the start-up for the move (including the current frame), which are displayed on screen for both clients.</td>
</tr>
<tr>
<td>Frame 8</td>
<td>The start-up animation has been performed on both clients to completion.</td>
</tr>
</tbody>
</table>

Unlike the lockstep protocol, there is no single standard for implementing rollback. Instead, developers must decide on their own implementations. One of the most important aspects of implementing a rollback solution is the method by which objects are serialized. The state of the game must be saved in order to be rolled back upon, and if an implementation is aiming to complete a rollback within a single frame, then serialization and recovery must be carried out as efficiently as possible. Serialized objects are used as savepoints, much like rollback functionality in databases. This means that, should there be a desynchronization between clients, the game will be able to roll back to any frame depending on the frequency of transmitted objects.

In Stallone’s GDC talk [13], he outlines their implementation of rollback in detail. Their solution introduced a system that supported up to 7 frames of rollback (Figure 7) with 3 frames of locally injected input latency prepended for processing purposes. The system uses parallel serialization, and only serializes mutable data, meaning if something has not changed during the rollback period, then it should not be saved. Stallone also highlights the importance of including data about object creation and destruction within their serialized objects as rollbacks will continuously move over creation and destruction edges, which will result in...
many more instances of objects being created and destroyed if handled “naively” [13]. Later, Stallone clarifies that this is not necessarily a bad thing, and that “sometimes the basic approach is exactly what you need” [13]. Stallone outlines two different methods of solving the issue of object creation and destruction in a rollback system, one being deferred object deletion, and the other being the standard delete and recreate. With deferred object deletion, the system can delay the deletion of an object to after the rollback period has completed, meaning that less data needs to be included in the serialized object, and so fewer resources are required to run the system efficiently.

Stallone states that throughout development, the team worked using a debug tool that simulated the full 7-frame rollback for every single frame, which essentially was the worst-case scenario for the game where the maximum amount of desynchronization is occurring. However, most players with a stable internet connection would only see a rollback of one to three frames at most when taking the initial input latency into account, meaning that the user experience was heavily improved [13].

When implemented optimally, rollback networking can simulate the feeling of true real-time online gameplay, much akin to two clients communicating locally. When implemented poorly, or in the presence of significant network delay, the results can be hugely detrimental to gameplay, with character avatars skipping and jittering as frequent rollbacks occur.

Input delay, i.e. a simulated delay between a player’s input and the resulting action on screen, is essential within a rollback implementation. When a client rolls back, the first frames of an opponent’s move animation will essentially be cut off and continue animating from some point during the animation. In an implementation with no injected input delay, clients would be continuously rolling back, which introduces jittery movement and perhaps even teleporting characters. Martin Mauve, in his research into maintaining consistency in interactive media, labels these discrepancies as “short term inconsistencies” that can be mitigated through the inclusion of “local lag” (input delay) [16]. These findings are also confirmed by Liang and Boustead, who state that local lag can be used to avoid state inconsistencies but is often insufficient unless conditions are optimal (no packet loss and low network latency) [17]. A further study by Mauve et al confirms that the combination of timewarp (rollback) and local lag can produce an optimal solution for mitigating network stresses and inconsistent states [18].

### 3.3.3. Prediction and GGPO

One of the most promising implementations of rollback networking, specifically in fighting games, is GGPO (Good Game, Peace Out). GGPO is a rollback networking framework developed by Tony Cannon, a computer engineer and long-time fighting game fan. It was developed specifically to attempt to hide network lag in fighting games, whilst preserving the responsiveness in local play. The framework uses speculative execution in the absence of actual opponent input data to essentially predict the inputs of a remote player [19]. This prediction is made by considering the previous actions of the remote player. This also helps remove the perceived input delay for local players, as they are not required to wait to receive opponent input data for rendering.
The most common form of prediction is that there are no inputs received from the remote player at all. This is also highlighted by Stallone when referencing their debugging tool that simulates a full 7-frame rollback. He comments that if there is no divergence between two clients, i.e. their states are synchronised, or if there are no inputs from a remote player, nothing should be happening within the simulation [13]. However, implementing an algorithm that only predicts that no inputs are received will also be a detriment to gameplay, as this will result in teleporting or stuttering avatars whenever there is a drop in received data.

The algorithm employed by GGPO is briefly touched upon by Cannon in his article: “The built-in prediction algorithm assumes future inputs will be identical to the inputs most recently received from the remote player” [19]. He elaborates with an example whereby the movement of a character can result in the screen moving left or right. With an incorrect prediction and therefore rollback, the result can be a jarring skip of the screen back to a point that is correct for both players. With a prediction that the moving character will continue to move in the same direction, most cases of skipping can be avoided.

GGPO’s prediction algorithm can be compared to optimistic concurrency control (OCC) strategies within databases. In the same way that remote player inputs are predicted based on previous inputs, OCC strategies assume that all transactions submitted to the database can be executed without conflicts. Each transaction is executed and only validated before they are fully committed. If there are conflicts at this stage, the changes are rolled back [20].

The rollback implementation outlined by Stallone is also very similar to GGPO in that the latency is hidden within the first frames of each move. It is even declared to be “GGPO-like” by NetherRealm themselves [21]. Therefore, it could be assumed that both implementations calculate a “frame advantage” for each client. A frame advantage simply determines by how many frames one client is ahead of the other, and thus calculating how many frames the rollback period should be. Cannon provides the following formula for determining the frame advantage of a player:

$$\text{Last Remote Frame} + \left( \frac{\text{Ping} \cdot \text{Frame Frequency}}{2} \right) - \text{Last Local Frame}$$

This formula estimates the current frame being rendered by the remote client by adding half the packet round-trip time (ping) to the last packet received from the remote client and subtracting the frame being rendered on the local machine. The result will be the number of frames that the remote player is ahead of the local player, and therefore the number of frames to roll back [19].

The prediction algorithms used in both Mortal Kombat X, Injustice 2 and GGPO can be argued to come under the general category of dead reckoning algorithms. Dead reckoning in video games is the prediction of an avatars position based on previously established variables such as velocity and direction, or perhaps in the case of above, based on player inputs. Stallone briefly notes that their rollback implementation uses dead reckoning for client-side prediction of player positions, and although he does not specify how exactly it is implemented, we can assume that the method of prediction is either similar, or the same as the GGPO implementation.
3.4 Summary

This literature review has explored the two main ways in which state is handled within peer-to-peer networking video games, with a special consideration towards fighting games in particular. Delay strategies implement a stop-and-wait protocol in which clients must receive and confirm inputs from all other clients before the simulation can progress. This is done through the lockstep protocol which employs a one-way hashing scheme for “committing” inputs, which are then confirmed with their plaintext counterparts. Lockstep has further optimizations including Asynchronous and Pipelined Lockstep which relax the waiting requirement by limiting waiting only to clients that are interacting directly with one another.

Rollback strategies allow for the local client to move as they wish without the simulation requiring the inputs for the remote player. Instead, the inputs of the remote client are predicted using some kind of prediction algorithm. In the case that the received inputs are different from the predicted inputs, the simulation rolls back to a previously established state and recalculated taking into account the correct inputs.

Both delay and rollback are commonly used in fighting games, although rollback seems to be the preferred method of handling client states by both Stallone [13] and Cannon [19]. This could confirm the prediction that participants in the case study would find the delay implementation to be more frustrating than rollback. Likewise, the preference also signals that rollback is more performant and fault tolerant than delay.
4 Case Study

4.1 Background

The case study compares different models of maintaining a consistent state between clients. This was done by using a test fighting game in which both delay-based networking and rollback-based networking was implemented and tested by real users. There has been a good amount of research into consistency models for video games, although most of this research was conducted using client-server architectures, and mostly purely focus on newly proposed models and protocols, whereas the aim of this case study was to strictly document how users perceive these continuous consistency models which were specifically implemented using a peer-to-peer architecture.

As briefly discussed in the literature review, Terrano and Bettner [10], as well as Claypool [11] both studied the effects of latency in real-time-strategy games. Both concluded that for these types of slower-paced games, the threshold for latency to cause issues with gameplay was significantly high (500-1000ms).

Pantel and Wolf [20] conducted a study on the effects of on the user experience of a multiplayer game. They analysed both commercially available racing games as well as their own test racing game. As expected, the results of their study show that as network delay increases, user experience degrades. This is shown both using subjective results (user feedback) as well as objective results in their measurements of lap times for each of their test users. Their study only tested the effects of delay/network latency.

A similar study was conducted by Savery and Graham [22]. User perception of state corrections was tested using a test game. The study found that users tend to prefer “smooth” corrections, i.e. interpolated positions of character avatars, as opposed to “warping” corrections where an avatar simply snaps into the correct position. The study also explored ways in which corrections could be hidden from the player, for example by providing distractions on screen. They found that unless the distraction shifted the player’s focus (i.e. the distraction was of little importance) then the player would still notice corrections [22].

Canfield [23] also measured the effects of latency on users, as well as documented different methodologies of state management within video games. Much akin to Savery and Graham [22] Canfield found that as network delay increases, user frustration also increases. Canfield’s game implementation used a client-server architecture, however it contained “peer-to-peer modifications” in that the server did not act as an authority regarding movement. It also gave one player an advantage over the other as they acted as the “server” in the simulation, meaning they were not subject to the effects of network delay. Canfield’s study, like Savery and Graham’s, was built upon by this case study as two different types of continuous consistency models were tested, with each having their own simulated network stresses for which users provided feedback.

4.2 Gameplay mechanics of a test fighting game

The player has access to directional input, and four buttons. Directional input is employed to move the character across the simulation space, characters may move forwards, backwards, jump, jump while moving either forwards or backwards, or duck. The characters are made to always face one another, the player designated to be “Player 1” begins facing right, “Player 2” begins facing left, therefore, “forward” motion is considered to always be a movement towards the opponent, and “backward” motion is, conversely, movement away from the opponent.
Orientation of the characters may change during gameplay (such as if one player jumps over another), these concepts, however, remain consistent.

Each one of the buttons corresponds to a combat move, each one with different characteristics. A move input triggers an animation on the player’s character, which is the only method at the player’s disposal to attack an opponent. Each animation is formed by a series of “frames”, individual states that are sequentially stepped to, after a fixed amount of time has elapsed. An animation frame may change the visual appearance of the character, but it also may contain two types of positional information: an array of “hurtboxes”, and an array of “hitboxes”. Each “box” consists of a pair of local 2D coordinates. If any of the hurtboxes in an animation intersects the opponent’s hitboxes, a hit is registered.

The player is not allowed to move as a combat move animation is playing, if they are in mid-air, they will conserve their momentum as the animation plays. Each animation consists of three stages, a series of “start-up” frames, that contain only hitboxes, followed by “active” frames, which contain a hurtbox and therefore are the first instance where the player can cause damage, and finally a series of “recovery” frames.

What follows after a hit is detected depends on the current state of the opponent, and this is illustrated in Figure 6. If the opponent is idle, the opponent’s health bar is reduced by the amount of damage of the move, the opponent is rendered unable to move for pre-set number of frames (this will now be referred to as “hitstun”), and a slight amount of horizontal velocity is applied (from now on referred to as “knockback”). If the opponent has their back against a wall, the “knockback” velocity is applied on the attacking player instead. The player may now also skip their move’s recovery frames by inputting another move, creating a series of moves that are chained together, or “combo”.

If the opponent is holding a backwards input at the time they were hit and were not in the middle of performing an action, then the move is “blocked”. A blocked move inflicts far reduced damage, causes a reduced amount of knockback and the opponent is unable to move for less frames than with a regular hit, potentially allowing for a retaliatory move during the player’s recovery frames.
Conversely, if the opponent is hit during the “recovery” or “start-up” portion of their moves, a “counter” is registered, and the player is rewarded with additional damage and hitstun, potentially allowing for longer combos.

These mechanics are intended to closely replicate those of modern fighting games such as Street Fighter or Mortal Kombat, and are intended to reward careful timing and execution, as well as quick reflexes. These two imperatives drive the need for efficient networking, as delays of merely a couple of frames could mean the difference between a move landing or not, or a counter hit being registered, thus enabling two different paths for the attacking player.

4.3 Implementation in Unity3D

4.3.1 Game simulation and State handling

As a game engine, Unity3D provides out-of-the-box solutions for peer-to-peer networking [24], input, rendering and supporting a game loop, that is, periodic execution of game logic. There is a need for the game’s state to be reversible, which is an important design consideration, as it means all state transitions must be tracked and stored, and it also means that future state must be entirely able to be inferred from the current one. As a result, floating point arithmetic could not be used for state calculations, as precision errors would accumulate over time, leading to potential desynchronization of the two simulations running remotely. Using integer arithmetic removes the possibility of numeric instability causing hit detection errors.

This has repercussions in the design of the game simulation: physics, such as projectile motion equations used for jump arcs, must have all their values precalculated so that motion matches exactly in both clients. It also implies that the axis-aligned bounding boxes used to check collisions must employ integers only. It should be noted that this is a requirement only for simulation logic, rendering and positioning of characters in 3D space can still employ floating point arithmetic, as these can be inferred from the state without mutating it.

Fighting game mechanics are designed with an immutable state in mind; the previous state and the current set of player inputs are what is required to calculate the current simulation state. The combination of both state and input at a particular point in time is henceforth referred to as a “game frame”. The game keeps track of it’s current frame as well as a buffer of all previous frames as a way to enable rollbacks. A rollback consists of replacing the current frame with a previous one, and then proceeding to recalculate all game frames using new inputs, until the current frame is reached.

Inputs are represented as a single byte, employing a bitmask. The fourth least significant bits each correspond to one of the four attack buttons (as more than one can be pressed at any point in time). The remaining four are used for directional input, described as integers from 1 to 9. As a mnemonic device, movement integers follow the layout of the numeric pad of a standard-sized keyboard, i.e. 9 corresponds to an up-right input, 4 to a left input.

A presumed initial state for the player must be entered as frame 0, so that the first player input is able to transform it. This initial state is identical for both player 1 and player 2; both players differing only in their orientation and initial X coordinate. The coordinate system used in the game has the middle of the play area as its origin. Both players start at a fixed distance from each other, facing one another. The play area also has fixed bounds at each edge, henceforth called “corners”.

24
The current player state is computed using bitmasks. The 4 least significant bits are used for movement states, and they may be in superposition. These movement states are: moving left, moving right, crouching and jumping. A player may only transition from their movement state if they are touching the ground. The 4 most significant bits are used for attack stats: the player is attacking, blocking, or is hit by an attack. The remaining bit is left unused in this simulation but may be used for special movement options, such as dashes. A player state of 0 is assumed to be the player standing idly, from now on referred to as being in a “neutral” state.

A circular buffer is employed for input delay (or input buffering). It is intended to be read from the middle of the array, yet otherwise operating as a queue, meaning that there is a delay between an input and its actual execution, as well as a queue where previous inputs remain until they cease being relevant for state calculations. The entire buffer, along with the player state is stored along the game frame.

The simulation maintains a record of every frame that it has received from the network, as well as every frame inputted by the player. Once a frame has been networked, the player is not allowed to change it and so the frame is then considered “committed”, and even in the case of rollbacks will not be altered.

While the simulation must run at a fixed rate of 60hz, the Unity game client can run at any frame rate above that. Simple characters, composed of primitive shapes, as seen in Figure 9, were employed as player characters, their movement, position, orientation and visual status were derived from the internal simulation state.

4.3.2 Networking

Following a traditional P2P approach, any game client running the simulation becomes a host by default. It can connect to another host and thus become another client, while the host employs a local client to see the simulation. All communication is done over a TCP socket, as delivery validation and packet order are important to guarantee synchronicity. Three types of messages are employed, all of which follow the same format, having a message ID, which is used internally to determine which callback and which handler to employ for its processing, and a message body consisting of a serialized binary representation of the data to be sent.

These messages are:

1. Set the current input buffer size so that it is consistent between both simulations (This message is also employed as an acknowledgement that the socket is open, and the simulation is loaded). The body contains the client’s latency in milliseconds, and the input buffer size is calculated by adding that latency to the local client’s latency and dividing it by the double the duration of a simulation frame, thus 30Hz. A constant is added to account for delays caused by engine processing in the local client, resulting in a minimum input delay of 5 frames.
2. Send X number of frames as a byte array, along with the frame count corresponding to the index of the first frame of the array. These are inserted into a list that tabulates all remote inputs in the local client. Through this list the user can infer if it must delay, or rollback,

3. Request a particular frame to be resent. This request returns the desired frame, plus a number of frames that follow it that equals the amount of input delay. This is used in the case of packet drop, when the next remote input frame to be read is detected to be null. Depending on the behaviour chosen, the simulation will either pause, or continue playing and rollback once it is received.

Network effects are simulated using a random number generator and a coroutine, as well as user entered settings which are visible upon first opening the game. As seen in Figure 10 “Artificial lag ms” refers to a delay in milliseconds added before sending any frame messages to simulate latency, “Packet Drop %” describes a fraction representing the probability of a message to not be sent at all, and “Input Buffer Size” determines the amount of frames on which to delay current player input, which can be set arbitrarily regardless of real latency.

Two configurations of the game were created. The default configuration is effectively an implementation of “delay” netcode, that is, the simulation will pause if no remote input frames are available. The other is rollback, where the game does not pause if a remote frame is available, but instead continues the simulation using the last available input as a placeholder, until the missing frame is received, thus setting the current frame, and simulating as quickly as possible, as fast as Unity allows and no longer bound to 60Hz, until the frame where the deviation began is reached.

The simulation outputs text files including a string representation of the game state, including player states and inputs per game frame, as well as a compilation of statistics, used for further analysis of the two techniques, such as the amount of time in both frames and milliseconds the player is forbidden from entering an input due to corrections caused by network effects, or the longest combo duration in the round.
5 Results

5.1 Literature Review

“How can a consistent state be maintained between two clients over a network, where no client has authority over another?”

The literature search explored two major ways in which state can be managed within non authoritative multiplayer video game implementations. Delay-based networking and Rollback-based networking are proven solutions that provide suitable state management systems when implemented in suitable types of games. Rollback networking caters to faster paced video games, with implementations seen in fighting games as described by Stallone [13] and Cannon [19]. Delay-based networking, although it can also be used in fast paced games, is more suited to games that do not require users to react quickly to other client actions, for example real-time-strategy games as outlined by Terrano and Bettner [10] and Claypool [11].

With regards to the ways in which states are specifically handled: delay-based networking ensures that all clients have sent and confirmed their inputs with one another, à la the lockstep protocol as outlined by Baughman and Levine [9]. Only when all commands have been received and verified can the game progress. This ensures that all clients have the same state at each turn. There are optimizations to the pure lockstep protocol [9][14], however all optimizations will eventually rely on a ‘lockstep mode’ to resolve the states of clients. In the event of desynchronization, either the whole simulation, or clients within a sphere of influence will simply wait until the issue is resolved. If the issue cannot be resolved, then it is up to the developer to implement their own countermeasure, for example, disconnecting all clients and cancelling the game.

Rollback networking does not guarantee that all clients are viewing the same simulation unlike delay networking. Instead, prediction algorithms are used to simulate the remote client’s inputs or movements depending on previously received data (dead reckoning). In the meantime, when remote client data is received, the simulation will roll back the remote client’s state to the point at which the data is received, simulate the received client data and resume. When desynchronization occurs, the remote client will most likely jitter and teleport (lag). These effects can be mitigated with the inclusion of simulated input delay or local lag[16][17][18], but in extreme cases, the effects of network latency will not be completely nullified, and so once again, it is up to the developer how countermeasures should be handled.

Therefore, to explicitly answer the research question with respect to video games, clients on a peer-to-peer network configuration can maintain a consistent state without an authoritative entity by implementing a delay-based solution or a rollback-based solution, where clients either wait for all inputs to be received, or predict other client inputs based on previous inputs and roll back to a previous state if inconsistencies appear.
5.2 Case Study

“How do different continuous consistency models compare to one another in terms of performance and failure tolerance with regards to user perception?”

To reiterate, the following predictions were made prior to the case study taken place:

1. Player inaccuracy would increase with latency and packet drop.
2. Round length will increase with latency and packet drop.
3. Delay networking will cause more frustration than rollback networking.
4. Rounds with lower latency will be felt as more responsive.

The results for question 1 in the questionnaire show a clear correlation between the ability for players to react and the amount of latency injected into the system. In fact, once latency values reached 100ms and above, the majority of participants strongly disagreed that they were able to react quickly to their opponents’ actions (see Figure 11).

Contrasted to the same question for the rollback tests in Figure 12, there is clearly a similar pattern in that participants felt they were unable to react quickly as latency values exceeded 100ms. Although, for the rollback implementation, participants’ disagreements are less pronounced, which supports the conclusion that rollback networking is better at dealing with network stresses. This is understandable, since the rollback simulation does not require the local player to wait for inputs of the remote player, but instead allows the local player to move freely.
Interestingly, the delay results for question 1 showed fewer agreements than expected for latency values of 50ms and below. 50ms of network latency is incredibly low and would be deemed as acceptable in any commercially available product. This outcome could be due to player experience, or perhaps the order in which stages were presented, where players would become more accustomed to the game as more stages were completed.

In the delay networking tests, participants’ answers to question 2 (with regards to responsiveness of the system) were found to be closely related to answers for question 4. In situations with high network latency and packet loss, delay networking implementations stop the simulation whilst inputs are received and confirmed by both players. Therefore, an assumption can be made that as latency increases, responsiveness decreases and the ability to react to the opponent player’s actions also decreases.

Figure 13 confirms this assumption in that participants felt the responsiveness of the system decrease as network latency increased. Compared to the answers for the rollback solution in Figure 15, most participants agreed that the simulation was responsive even with latency values of 100ms, with strong disagreement only for values of 200ms. This again supports the claim that rollback solutions handle network stresses more efficiently than delay solutions.

The results also revealed that participants could clearly feel the effects of network latency in both delay and rollback solutions. In their answers to question 3, participants all agreed that the network connection was unstable at values of 100ms and above in the delay simulation (see Figure 15). These results are much less pronounced in the rollback tests, as most users only determined the connection to be unstable at latency values of 200ms (see Figure 16).
Interestingly, in both delay and rollback tests, participants determined the network connection to be stable in stages 4 (no latency, 10% packet loss) and 9 (no latency, 20% packet loss). Likewise, for questions 1 and 2, the responsiveness of the system and their ability to react were also mostly strongly positive. This outcome is most likely due to the inclusion of an input buffer in both implementations.

The input buffer refers to the number of frames the local player’s current action is delayed, and the number of frames sent per packet per frame. It represents a trade-off between user input reception and perceived connection quality, as a higher amount of input buffering will result in higher perceived input latency, but also increases the number of remote frames the simulation has available to process. Since an input packet is sent every frame, and they contain several “future” inputs, for the simulation to run out of frames it would require a succession of packet drops that is greater than the amount of input frames buffered.

The input buffer scales with the amount of round-trip latency in the connection, with a minimum of 5 frames. As these stages did not add artificial latency, the input buffer size was therefore 5 frames. This meant that the event of a packet being dropped, with 20% probability, would have to be repeated at least 5 times in direct succession for it to be noticeable to the player, a joint probability of 0.00032%. In a match with 3200 input frames, this is estimated to occur only once.

The inclusion of the input buffer, and the clear results showing that participants could not perceive the effects of especially high packet loss, provides further proof that “local lag” helps mitigate the effects of network stresses, both for delay and rollback solutions.
Participants were asked to grade their performance after each stage of testing. These results, both for rollback and delay, loosely show that for stages with high input latency (100ms-200ms) participants felt they performed worse than for stages with lower latency (see Figures 17 and 18).

![Figure 17: Participant responses to Q4 for delay tests](image)

![Figure 18: Participant responses to Q4 for rollback tests](image)

The results show that for the delay tests, most users did not play well when latency values were 100ms or higher, whereas with rollback, these results were only mirrored in stages with 200ms latency. Participants also performed noticeably better in the rollback simulation, although this could also be a result of participants becoming acclimated to the game.

Although there was no question on the questionnaire regarding a participant’s frustration after each stage of testing, the fact that most participants graded their performance much lower in stages with high latency could somewhat confirm that players would be more frustrated in these less performant stages.

Before the main testing phase started, the participants were also asked to rate their experience with video games on a scale from 1 to 5, with 1 being not experienced at all, and 5 being very experienced. The results were somewhat skewed towards the participants being more
experienced, but there were still participants who had little to no experience. Figure 19 shows the distribution of participants based on their experience.

When all responses were compiled for each question with experience considered, the only data that was somewhat variable was that of question 4, where participants graded their own performance. Otherwise, it was clear that regardless of experience, participants could still tell if the simulation was responsive and if the connection was unstable for both delay and rollback solutions. Figures 20 and 21 show these results. The remaining questions in this format are available in Appendix 1.

![Figure 19: Video games experience distribution among participants](image1)

![Figure 20: Delay Q5 answers with experience considered](image2)
Despite being able to clearly tell when the simulation was responsive and when the network was unstable, participants of a lower experience level mostly graded themselves much lower than those with more experience. During the delay tests, those with experience 1 and 2 were nearly always at least two grades below those with experience 5. The same is true for the rollback tests, although on average, participants with less experience graded themselves higher during these tests. This could also confirm that rollback solutions provide better performance than their delay counterparts, especially considering that, in the rollback tests, players with less experience graded themselves higher in stages with latency values of 100ms than in the delay tests.

In summary, the following conclusions, and therefore answers to the proposed research question, can be drawn from the analysis of user feedback:

1. Rollback-based networking solutions provide better fault tolerance than delay-based networking in faster paced video games. This can be seen by participants’ agreement of responsiveness in stages with up to and including 100ms of injected latency with rollback, as opposed to just 50ms for delay.

2. Rollback-based networking is also more performant than delay-based networking when under network stresses for the same reason. In addition to this, participants mostly only detected unstable network conditions during tests of 200ms for rollback, but for delay, participants determined that all stages with 100ms latency and above were unstable.

3. When using an input buffer to induce “local lag”, the effects of high packet loss can be sufficiently mitigated in low latency scenarios, which can be seen by all participants’ positive answers for all stages with only packet loss present.

Figure 21: Rollback Q5 answers with experience considered
6 Discussion

The results of the case study are much in line with the findings of the literature review. In the review, whilst discussing both delay and rollback networking, it was strongly suggested that rollback solutions were more suited to faster-paced video games [13]. This was further proven in the case study, where test users found that rollback provided a more responsive environment with higher tolerance to network latency and packet loss. Likewise, although this was not necessarily one of the main tests for the case study, it was also further confirmed that the use of local lag in the form of an input buffer can effectively mitigate the effects of packet loss, as suggested by Mauve [16][18] and Liang et al. [17].

There is no doubt that more testing should be carried out with regards to continuous consistency models in peer-to-peer networked video games. For this case study, had there been a larger number of participants, it is entirely possible that a different conclusion could emerge, or maybe the current results could be proven definitively. However, as it stands, the implications of the case study are still significant in that there was a clear perceived performance difference between the delay and rollback solutions.

Perhaps a significant reason why participants noted a performance difference between the two models is the noticeability of a rollback as opposed to a delay. For example, a user would easily be able to observe the entire simulation freezing for 4 frames (67ms) as opposed to only the remote player avatar rolling back 4 frames. In a fighting game, with only two characters on screen at one time, the latency in a GGPO-based rollback implementation is only observable in the remote player character. It is essentially localized, whereas the whole simulation is affected by waiting periods in a delay implementation.

In addition, rollback networking allows for correction of the current state to be performed far faster than delay. So long as hardware allows for the simulation thread to execute faster than 60Hz, the rollback simulation can be unlocked so that it executes faster than regular gameplay, as it is not constrained by the player input thread since all data is already available. Perhaps the reduction in time where the player is not allowed to make a new input results in the difference in perception.

Rollback does also by design require fewer packets to be sent back and forth, as it can differentiate between a frame message being delayed and a frame message being lost. Delay networking must always request frames that are missing, as the only alternative on a missing frame is to request it. Rollback can choose to only request frames once it is known that they have been dropped as opposed to delayed.

Rollback, while appearing as a net positive, does have some drawbacks regarding player perception. Players observed occasional unintended behaviour, such as sounds playing twice due to the simulation repeating a sound trigger on a rollback. While these effects can be mitigated with careful game design, it introduces additional user experience considerations, which lie beyond the scope of this study.

Most fighting games still employ delay-based strategies, with very few titles opting for a rollback strategy. This is arguably simply due to the fact that rollback models, and specifically the GGPO implementation, are still relatively new within the fighting game space. Therefore, it is possible that the results of this study and others like it could provide the proof needed for developers to eventually move towards the more performant rollback model.
The fact that users can clearly tell the difference in performance between the two models could also mean that these technologies could be beneficial when implemented elsewhere. Cloud-based gaming services are now once again entering into the video game space, with both Google’s announcement of their Stadia platform [25] as well as Microsoft and Sony’s announcement of their cloud gaming and AI partnership [26]. Cloud gaming services traditionally use delay-based strategies for streaming client inputs to the host system, but a rollback solution could be explored to potentially aid in responsiveness.
7 Conclusion

7.1 Summary

This work has explored ways in which continuous consistency models can be used to maintain states within peer-to-peer networked video games and analysed how these state management models compare to one another in real life implementations. Through a literature study, it was determined that the two most common strategies for managing states in peer-to-peer implementations were either employing a delay-based strategy, or a rollback-based strategy. Delay strategies required all clients to wait for all inputs to be received and confirmed before the simulation can move forward (lockstep protocol [9]). Rollback strategies ensure state synchronisation by predicting the actions of a remote player using some dead reckoning algorithm, and then “rolling back” remote players to a confirmed state once remote inputs have been received.

The literature review determined that delay-based networking was more suited to games that were not especially time-critical, such as real-time strategy games where there is a higher tolerance for latency. For faster paced games like fighting games, rollback-based strategies were much preferred due to the natural responsiveness created when clients are not required to wait for one another’s inputs before continuing the simulation. This was confirmed by the case study, where user responses showed that the rollback solution provided suitable responsiveness in all stages with latency below 200ms, whereas the delay solution was only responsive in stages with latency of up to 50ms. This showed that the rollback solution was both more fault tolerant and performant than the delay solution.

7.2 Ethics, Sustainable Development and Societal Aspects

The choice of state management models within video games as a whole is not trivial. As Stallone’s [13] account has shown, studios will carefully consider each implementation, and if necessary, completely overhaul their current implementation for another. This is not a quick and easy task. With the results gained from this work, there is a clear distinction between both delay and rollback strategies in terms of fault tolerance and performance. Perhaps studies such as this would both help studios make these decisions more quickly and contribute towards a smoother development process and prevent the need for drastic and expensive changes to be made very late into a game’s lifecycle.

Fighting games are also cementing their place within the global Esports industry [27]. Developers are creating tournament circuits for their own games, with notable examples including the Capcom Pro Tour and the Tekken World Tour, each with prize pools in the hundreds of thousands of dollars [28][29]. Within the fighting game community, there is somewhat of a stigma towards online play in that the effects of network stresses reduce the legitimacy of a player’s skill. High level tournaments are always played locally, where players from around the world will attend and are generally where players will wish to prove themselves. With more clearly defined decisions with regards to network strategies, and the success of GGPO in simulating offline-feeling play over an online connection, the stigma of online matches for fighting games could potentially be withered over time as more development studios embrace the supposedly more performant networking consistency model. This could lead to studios bringing their circuits
online, allowing for all players to participate, removing the often costly requirement of physically being at a tournament venue.

7.3 Future Work

The results of this work could be further expanded by both increasing the number of participants to give feedback, as well as possibly including analysis on the effects of age and/or gender on user perception. The participants for the case study were all young adults, and so there was not a sufficient spread of age to justify deeper analysis on the effects of age on user perception, however this could be potentially be a useful study as the average age of people playing video games slowly rises. Likewise, only three out of ten participants were women, each of whom had little experience with video games. Therefore, it was much more likely that any differences between their results and others was due to them having less experience rather than their gender.

Future work could include tests using implementations of games in other genres such as racing games, shooters, Multiplayer Online Battle Arena (MOBA) games etc. Other genres each have their own requirements, and therefore the optimal networking/state management system could be very different to the systems designed for this work.

The use of continuous consistency models in real time applications could potentially be used in the field of cloud gaming. Games designed with these techniques could guarantee that no inputs are lost and that the game experience remains consistent for multiplayer games played over streaming services.
8 References


9 Appendices

Appendix 1: Experience Based Questionnaire Results

Delay Tests

Delay - Q1: I could react to my opponent's actions quickly
P1 (1xp) = Participant 1 (Experience 1)

Delay - Q2: The game was responsive to my inputs
P1 (1xp) = Participant 1 (Experience 1)
Delay - Q3. The network connection was unstable
P1 (1xp) = Participant 1 (Experience 1)
Rollback Tests

Rollback - Q1: I could react to my opponent’s actions quickly
P1 (1xp) = Participant 1 (Experience 1)

Rollback - Q2: The game was responsive to my inputs
P1 (1xp) = Participant 1 (Experience 1)
Appendix 2: Source Code

Source code is available at: https://github.com/thefnox/ThesisProject

NOTE: Please consult the README for instructions to run.