Cheat Prevention in Peer-to-Peer Multiplayer Games: Application of the Raft Consensus Algorithm

Jan Wichmann

11127222

jan-wichmann@live.com

Written in the summer term 2021

for BA Digital Games

at Cologne Game Lab / TH Köln

supervised by Prof. Dr. Roland Klemke

submitted on 1/7/2021.

1. Abstract

The use of consensus algorithms to protect distributed systems against possible failures is already well-established. As the system agrees to work in union it maintains operations even while some of its components may malfunction. Thus far, the potential of this protective measure has yet to be considered as cheat protection in peer-to-peer multiplayer games. This paper aims to evaluate the eligibility of the consensus algorithm Raft to prevent cheating in peer-to-peer multiplayer games. While Raft cannot be utilized as a countermeasure against cheating directly due to the lack of essential safety properties, it can still serve as a benchmark for other Raft-based consensus algorithms that do meet the necessary requirements. The feasibility of those algorithms in the context of peer-to-peer multiplayer games is evaluated by comparing the additional communication delay that Raft imposes on the system with game genre specific latency thresholds. The thresholds mark the network latency at which a player's performance starts to drop so significantly, that the game becomes barely playable anymore. Raft adds a great communication delay to games in a peer-to-peer networks, rendering it useless as cheat protection for any First-person shooter games and Role-playing games, at least under non-optimal network conditions. The playing experience in Real-time strategy games does not suffer as drastically from high latency as in other genres, therefore Raft-based consensus algorithms can be utilized as cheat prevention in these games if they meet the necessary safety properties.

Table of Contents

1. Abstract ii

2. List of Tables v

3. List of Figures v

4. Introduction 6

5. Motivation 7

6. Multiplayer Games 8

6.1. Player Actions in Multiplayer Games 8

6.2. Game Network Architecture 11

6.2.1. Server-Client 12

6.2.2. Peer-to-Peer 12

7. Cheating in Multiplayer Games 13

7.1. Motivation 14

7.2. Classification 15

7.2.1. Exploiting misplaced trust 16

7.2.2. Collusion 16

7.2.3. Denying service to peer players 17

7.2.4. Exploiting lack of secrecy 17

7.2.5. Compromising game servers 18

7.2.6. Internal misuse 18

8. Consensus Algorithms 18

8.1. Fault Tolerance 19

8.2. Byzantine Failure 20

8.3. State machine Replication 20

8.4. Application 22

9. The Raft Consensus Algorithm 23

9.1. The Algorithm 24
9.1.1. Leader Election 25
9.1.2. Log replication 26
10. The Raft Consensus Algorithm in Multiplayer Games 27
10.1. Multiplayer Game as Replicated State Machine 27
10.2. Byzantine Failure in Raft 29
10.3. Validation-Based Byzantine Failure Tolerant Raft 30
10.4. Practicality of Raft as Cheat Prevention in Multiplayer Games 32
11. Conclusion 36
12. References 37
13. Statement of Originality 39

2. List of Tables

Table 1. Latency threshold in multiplayer games, Table, “Latency and online games”, (Claypool and Claypool 2006) 35

3. List of Figures

Figure 1. Player actions relative to deadline and precision, Figure, “Figure 1”, (Claypool and Claypool 2006) 10

Figure 2. Player performance relative to network latency, Figure, “Figure 3”, (Claypool and Claypool 2006) 11

Figure 3. Time per Commit in a Raft Cluster with different number of members and varying simulated latency. 33

Figure 4. Time per Election in a Raft Cluster with different number of members and varying simulated latency. 34

4. Introduction

Online multiplayer games are amongst the most popular and top grossing titles on the video game market (SuperData 2020). As the Entertainment Software Association states in their 2020 report, 65% of players are looking to connect with others through games. Most time is spent playing online, but also together in person (ESA 2020). Therefore, supporting the ability to play together with friends and others has become an important feature for most video games. Even games that mostly focus on a single player experience often include multiplayer or online features to some extent. However, with these games getting more and more connected through multiplayer capabilities also comes the potential for cheating. While there is a way to hold cheaters accountable when playing games against family and friends online or others in person, there is no way to do so when playing against strangers online. This disconnect between action and consequence as part of the anonymity of the internet can inspire players to fall back on cheating when playing online games. For game developers cheating is a major concern, especially for those games that rely on customer retention as players are less likely to continue playing a game where they encounter cheating frequently (DeLap, et al. 2004). Therefore, many game developers have opted for using a server-client architecture that addresses cheating from an infrastructural standpoint, which has then become the standard for most online multiplayer games (Webb and Soh 2007). However, the enormous financial investment that comes with deploying such an infrastructure can be such a burden for smaller developers that it discourages them to create multiplayer games all along. The ideal solution for small game developers with less budget would be to build games on top of a peer-to-peer infrastructure to remain scalability while also saving on costly server hardware and hosting services. The issue with a game running on a peer-to-peer infrastructure however is the very fundamental idea of giving up the control of the game to the participating players and therefore giving way to cheating again. There already are proven measures against some forms of cheating in peer-to-peer games such as the Lockstep

protocol (Baughman, Liberatore and Levine 2007). Looking at how the use of consensus algorithms has become a well-established technique to eliminate irregularities in other peer-to-peer systems such as distributed databases, the focus of this work is to evaluate if those algorithms can also be applied to peer-to-peer multiplayer games to prevent cheating. At first multiplayer games and popular genres with their specific player requirements will be described, while taking into account how the players' ability to act is effected by network latency. Next, a taxonomy on cheating in online games and a classification of relevant cheats is presented. In the following chapter consensus algorithms and their application will be discussed and later the Raft consensus algorithm is described in more detail. Finally, the last section will cover the evaluation of the application of Raft as a protective measure against cheating.

5. Motivation

As stated previously, the development of multiplayer games is a resource intensive investment. Not only does it take great efforts in terms of development time to create the game itself, but the technical complexity of online multiplayer adds an additional layer to it. While that time investment is a problem in and of itself, an additional problem for smaller developers becomes the money required to acquire and maintain the necessary infrastructure. With recent developments in server providing services solutions such as virtual servers, infrastructure can scale according to the needs of the developer. Still, it requires a monetary investment up front for players that might not even play the game. With a peer-to-peer architecture there is very high scalability since the players' machines are incorporated into the infrastructure. The system therefore has little to no running cost for the developer. The possibilities of peer-to-peer networks have been showcased in the recent uprise of cryptocurrencies such as Bitcoin (Nakamoto 2008). The market capitalization of the entire cryptocurrency market was estimated to be around 2.56 trillion US Dollars on May 5th, 2021. Compared to the around 75 million US Dollars 5 years before that, on May 5th, 2016, the market capitalization has increased by approximately 34133% in that time span

(CoinMarketCap 2021). This is a monetary evaluation of only that specific application of peer-to-peer networks, but it gives an idea of how much potential this technology has, if it is utilized in the right context. Another infrastructural benefit for a game running on peer-to-peer architecture is the decentralized nature of it, as there is no single point of failure. While other games relying on game servers might experience outage due to server issues or even malicious attacks on the infrastructure, games running on peer-to-peer networks are not effected by it. Also, this allows for a game to still be played even after a developer has abandoned the support, giving players more control over the product they purchased.

6. Multiplayer Games

Multiplayer games are digital games that provide the ability to play together with other players. The games can be played on the same device with multiple input controllers often referred to as local multiplayer or split screen multiplayer. However, the focus of this work lies on online multiplayer games, so games played over the internet. Those games still could be played in a similar fashion to local multiplayer games by gathering all players in the same physical location. Then each player plays on their own device and own input controller but in the same local area network (LAN).

6.1. Player Actions in Multiplayer Games

Multiplayer is a feature that can be found in almost any digital game today, across all genres. Each genre offers its own challenges to the player and therefore has its own requirements in the context of multiplayer. To get a better understanding of some of these requirements this chapter will outline the most popular game genres and how the players actions are affected by network latency.

Claypool and Claypool concluded that different kinds of player actions can be more sensitive or less sensitive to network latency (Claypool and Claypool 2006). They assign actions two distinct properties: precision and deadline. They are used to determine how sensitive an action is to higher latencies. The precision refers to the amount of accuracy the action requires, while the deadline represents the timespan between the start of an action and the final result (Claypool and Claypool 2006). The mentioned properties are not just inherit to the action itself, but are also determined by the interaction model and the perspective through which the player perceives the game (Claypool and Claypool 2006). They point out that the interaction usually either happens through an avatar, where the player's actions are limited to a character and its surroundings or through an all seeing and omnipresent entity that can impose its influence onto a larger set of the game world. The perspective, which is the way how a player perceives the game on their screen, can be different across different interaction models (Claypool and Claypool 2006). For example, a game with an avatar-interaction model might have a first-person or third-person perspective. To give actual examples, some popular game genres will now be outlined in more detail, including player actions.

First-Person Shooter

A First-Person Shooter (FPS) game focuses on an immersive and often reaction-based experience. It requires the player to perform quick actions and utilize precise aim. Many FPS games require the player to be vigilant for visual and audio cues to estimate the position of an enemy and therefore gain important advantage over them. As the name suggests, the game is perceived in first-person, so from the view of the player's avatar. Player actions are usually aiming, shooting moving targets or enemies, and traversing the game world by foot or in different vehicles.

Role-Playing Game

A Role-Playing Game (RPG) focuses on character development and customization. It often requires the player to invest a great amount of time into upgrading their character. Equipped items and apparel increase the character's ability to block and deal damage. Increasing the level on a character unlocks new

abilities or improves their effectiveness. Most common for RPGs is the third-person perspective where the player can see their own character. The player interacts with the game world through their avatar. Common forms of actions are either wandering around to explore, collect items and interact with other characters or fighting enemies through melee combat or some form of spells.

Real-Time Strategy

A Real-Time Strategy (RTS) game focuses on tactical finesse and management skills. It is vital to use information to your advantage and make decisions based on them. It requires the player to make long term plans, but also be able to adapt them quickly. The player actions mostly evolve around commanding units and placing buildings through an omnipresent entity, often viewing the game through a top-down or bird-eye view.

Claypool and Claypool have put together a figure that visualizes the precision and deadline properties of player actions across different interaction models and game perspectives (Figure 1) (Claypool and Claypool 2006). As seen in Figure 1, any games perceived through an avatar in first-person require a lot of precision and actions are mostly short in duration. Avatar-based third-person games are usually more forgiving in terms of precision and actions take longer to perform. Any actions in omnipresent games are rather long in duration, as they yield no immediate results.

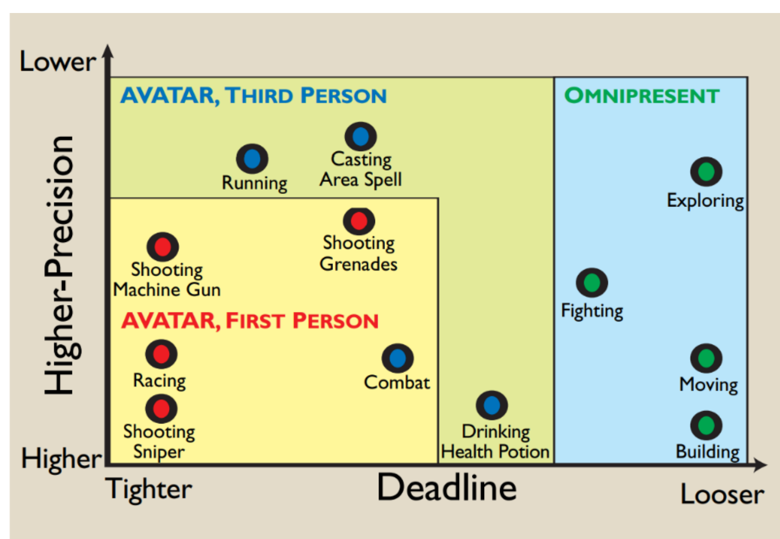

Figure 1. Player actions relative to deadline and precision

According to Claypool and Claypool, there have been various studies about the effect of network latency on the performance of the player. The experiments measured key gameplay metrics of different game genres under varying network conditions (Claypool and Claypool 2006). In Figure 2 these performances are grouped into game genres and put into relation with network latency. In the chart, a performance of 1 describes the best performance and 0 the worst (Claypool and Claypool 2006).

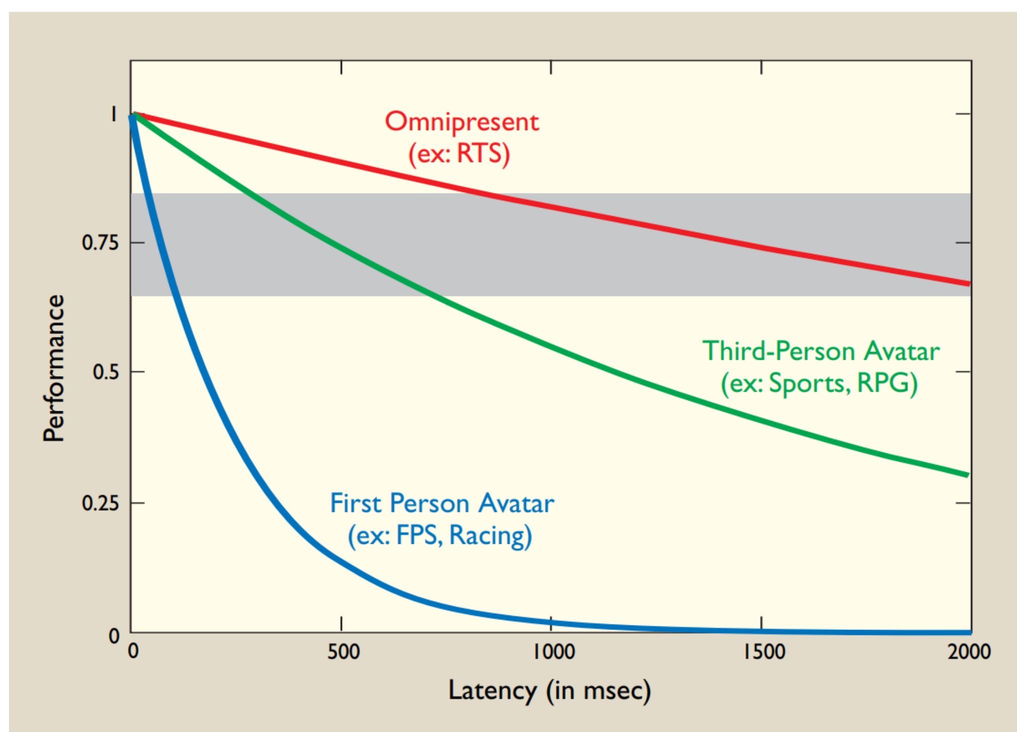

Figure 2. Player performance relative to network latency

6.2. Game Network Architecture

Network architecture describes the arrangement of the infrastructure responsible for providing all functionality of a multiplayer game. Usually, a player is referred to as a client and the entities responsible for supplying and storing information are called servers. Depending on the size of the game there might be the need of multiple servers to work in parallel to handle a great number of concurrent players. Also, multiplayer games might have different sets of services with their respective sets of data that are handled by different servers. A login server might

handle account registration, login, and all account administrative functions while the actual game server handles the real time communication with clients and updating the game world accordingly.

6.2.1. Server-Client

The Server-Client architecture is the traditional and well-established infrastructure used for most internet-based services where a single server or multitude of servers are utilized for information storage and processing. The developers have full control over the servers and can deploy security measurements to make sure that information is safe and cannot be wrongfully altered (Webb and Soh 2007). Still, a server-client architecture is highly centralized and therefore prone to a multitude of network attacks. Once the server is unreachable the entire network and therefore its services become unavailable. Outage can be mitigated through the use of multiple servers and backups to make the system fault tolerant, allowing some of the system's components to malfunction without loss of data or network availability. All these features come at a great monetary cost since it requires to either setup one's own server or rent one from a hosting provider (Kabus, et al. 2005). Building such a server needs a great investment up front and requires proper technical knowledge to setup and maintain. Renting servers allows for partitioning of the computing power over multiple physical locations for better connectivity for the end user. In addition, the technical supervision is done by the hosting provider. Of course, those conveniences come at a greater cost than just the hardware itself since the services and taxes are added.

6.2.2. Peer-to-Peer

The Peer-to-Peer architecture consists of multiple peers that act as both servers and clients at once. Each member of the peer-to-peer network is responsible for information storage and processing. Because of the decentralized nature of peer-to-peer, the network has no single point of failure. Trying to bring down the

network requires an attacker to shut down all its members. The developer can deploy a service running on a peer-to-peer network with no running cost since the network's available resources grow with the amount of members (Kabus, et al. 2005). These natural traits of decentralization and scalability come at the cost of control over the network. With no central server, authority is now distributed over the entire network. By demanding processing or computational power from its members, the network therefore supplies them with power in the decision-making process. Security is a great concern in peer-to-peer networks since members are directly communicating with each other. While it might be difficult to attack the entire network, individual members might be targeted since their IP addresses are exposed. In general, the problem in public peer-to-peer networks is that potentially malicious members can join the network and corrupt its state.

7. Cheating in Multiplayer Games

In the following the definition of cheating in the context of multiplayer games is used from the work of Yan and Randell, that states:

> *"Any behavior that a player uses to gain an advantage over his peer players or achieve a target in an online game is cheating if, according to the game rules or at the discretion of the game operator (that is, the game service provider, who is not necessarily the developer of the game), the advantage or the target is one that the player is not supposed to have achieved."*
>
> (Yan and Randell 2009)

Since game developers and publishers are making their game and connected services public it is often a constant cat and mouse game between cheaters trying to find new ways to exploit the game and developers trying to mitigate against it. Cheat developers get to analyze the existing infrastructure and security measurements with complete access to the source code running on clients. This

will always give the game developer and publisher a disadvantage since they must react to a cheat while it is already actively damaging their game and service.

7.1. Motivation

The motivation to cheat in multiplayer games can greatly vary between different games, depending on the avenues of how players can interact with each other and how game mechanics influence each other. Three goals for cheating have been formulated: defeat the competitor or avoid failure, get higher game level, and obtain economic interest (Lan, Zhang and Xu 2009). In some instances, a cheater's immediate goal might differ from the overall motivation. Games can have ranking systems that reward winning against opponents, here a high rank within the game is often considered a prestige and is therefore very desirable. Outside of the game's eco-system the account connected to the cheater can then be sold for real-world currency. The same concept applies to games where a high level or high-level character enables a richer gameplay experience and/or requires intense time investment. So, in the end, the motivation to win and level up can be goals in and of themselves or are just intermediate goals along the way of obtaining real-world economic interest.

In Addition, griefing shall be added as a goal as well. A griefer is defined as:

> *"A player who derives his/her enjoyment not from playing the game, but from performing actions that detract from the enjoyment of the game by other players."*
>
> (Mulligan and Patrovsky 2003)

This means a player might very well cheat in a multiplayer game just to see other players' experiences suffer due to their unfair advantage. While all players that grief are certainly not cheaters, it does make it a lot easier for them to do so if

they cheat, since now other players have virtually no way of defending themselves by just sticking to actions allowed in the rulebook.

7.2. Classification

To understand how players cheat in online games it is important to define the different well-known kinds of cheats and exploits. The work of Yan and Randell tries to shine a light onto this growing field of online security and aims to create a standard for the classification of online cheats. Their systematic taxonomy characterizes cheats by what vulnerability is being exploited, what consequences the breach has for the system, and what parties are involved in the cheating process (Yan and Randell 2009). As a result, they have come up with 15 different categories of cheats:

- Exploiting misplaced trust
- Collusion
- Abusing Game Procedure
- Cheating related to virtual assets
- Exploiting machine intelligence
- Modifying client infrastructure
- Denying service to peer players
- Timing cheating
- Compromising passwords
- Exploiting lack of secrecy
- Exploiting lack of authentication
- Exploiting a bug or loophole
- Compromising game servers
- Internal misuse
- Social engineering

(Yan and Randell 2009)

A lot of the cheats listed are revolving around exploits outside of the game and its direct infrastructure. Therefore, these cheats will not be discussed further, as they are not the focus of this paper. Going forward the emphasis will lie

specifically on the categories of cheats that Yan and Randell named: exploiting misplaced trust, collusion, denying service to peer players, exploiting lack of secrecy, compromising game servers and internal misuse (Yan and Randell 2009). In the following these cheats and those relevant to the integrity of the game and its direct infrastructure will be briefly described to create a basis of understanding for the remainder of this paper.

7.2.1. Exploiting misplaced trust

Usually, any information coming from a player's computer cannot be trusted. Since the client side is under complete control of the player, information could have either been tampered with or outright faked entirely. This means the client program itself might have been modified to send fraudulent messages to the server or other players. Even if the game client itself remains untouched an external program or proxy might interfere the outgoing messages and modify their contents. Not only is the information coming from a client a potential risk, but also the information going to a client as well. The client program can be altered to such a degree that information usually hidden from the player is now visible and therefore gives an unfair advantage. Again, even if the game client remains intact, an external program could easily read critical information from the computer's memory and display it as an overlay on top of the game (Yan and Randell 2009).

7.2.2. Collusion

Players can work together at their own benefit although they might represent opposing parties in the game itself. To communicate the colluding players could be using third party tools to plan and exchange information. The players might choose to not attack each other, even though they are not playing on the same team. An example of this is the so-called "win trading" seen in the real-time strategy game StarCraft. Here two cheaters worked together to gain advantage in a competitive ranking system. The cheaters would take turns by letting each

other win, giving them victory points essentially for nothing. This would increase their ranks in the competitive ladder and allow them to climb to a top ranking without playing the game as intended (Yan and Randell 2009).

7.2.3. Denying service to peer players

In most online games that utilize a server-client structure, where the server is a dedicated machine and not one of the players, denying service to a player is not really feasible unless players are directly connected to each other through a Voice over IP (VOIP) service that might run over peer-to-peer and expose their IPs. However, in a peer-to-peer network the potential of an attack on one of the players is a lot higher since all IPs of participating players are available. To deny the service to a player one or multiple attackers can flood their opponent's network connection to drive up their response time and potentially force-kick them (Yan and Randell 2009). A lot of games will kick a player if their ping exceeds a certain threshold. This mechanic is often used to perverse an enjoyable playing experience for other players in a lobby. In case of a server-client structure being used, where the server is running on one of the players' machines or is under their direct control, denying service to specific player is even more accessible since the server can just outright stop message from being sent to or received from the targeted player.

7.2.4. Exploiting lack of secrecy

The communication between players can also be a gateway for cheating. When messages are not encrypted and sent in plaintext they might be intercepted and wrongfully altered (Yan and Randell 2009). This can be a particular issue in peer-to-peer networks when the communication between two players cannot be established directly. For example, if players A and B are only connected with each other through player C then any messages that are sent between the two in plaintext might have been modified by player C for their own benefit.

7.2.5. Compromising game servers

The same way how cheaters might modify the client program, they might also modify the server program. This issue is especially of importance in peer-to-peer networks where essentially every member of the network is a server and a client at the same time. In traditional server-client architectures the access to a game server might not be as trivial then but is still possible if there is a lack of security measurements.

7.2.6. Internal misuse

Internal misuse describes a type of cheat where an administrator with special privileges abuses their position of power to gain advantage in the game. This can be any altering of the game world such as giving their own player powerful items or abilities (Yan and Randell 2009). This kind of power abuse is possible mostly on centralized systems such as server-client architectures because there is one single authority that dictates the game state. However, on a decentralized system such as a peer-to-peer network there is no central authority, therefore the entire network needs to be convinced that a certain action happened. Decentralization can be a great option to mitigate the abuse of a powerful authority.

8. Consensus Algorithms

Consensus Algorithms are utilized in computer networks made up of individual machines to allow them to operate as a whole unit, even if some of those machines experience a failure (Ongaro and Ousterhout 2014). The most established consensus algorithm till this day remains to be Paxos, developed by Leslie Lamport in 1998. The algorithm is named after the Greek island of Paxos. According to Lamport, many hundred years ago the ancient theocracy was

abolished, and a parliamentary form of government was established on the island instead (Lamport, The Part-Time Parliament 1998). He describes the new government as a “part-time parliament” since some of its members could not dedicate all their time to politics. This meant that the parliament had to remain functional even in the absence of some of its legislators (Lamport, The Part-Time Parliament 1998). Lamport draws parallels between the problem of ancient Paxos and the one of today’s fault-tolerant distributed systems. He suggests the legislator be the equivalent to a process that is part of a distributed system which should, just as the parliament, continue to function even if one or few of the processes fail.

8.1. Fault Tolerance

Fault tolerance describes a system’s ability to continue functioning even in the case of a failure within its components (Johnson 1984). Johnson describes failures in digital systems as a set of categories: “specification errors, implementation errors, external disturbance, and random component failure” (Johnson 1984). Specification errors are mistakes made in the planning and design of the hardware and software of a system, according to Johnson. Next, he defines implementation errors as mistakes made during the construction and development of a system’s hardware and software. The last two categories are only related to the hardware and are both not in direct control of the system’s operator. Both could be considered rather external faults, as Johnson considers external disturbances to be any natural hazards such as radiation and electromagnetic interference and component failure to be mistakes done by the hardware manufacturer during production or component burn outs (Johnson 1984). Ultimately, such a fault in the system could cause unavailability or corruption of resources. These failures mentioned are mostly referred to as crash failures, in the next section another relevant form of failure will be discussed.

8.2. Byzantine Failure

The Byzantine Failure is a term coined by Lesie Lamport in 1982 (Lamport, Shostak and Pease, The Byzantine Generals Problem 1982). It describes a specific behavior of faulty components that continue to operate but propagate false or contradicting information. This component might just malfunction because of failures mentioned in the previous section and continues to share the information that it considers to be correct with other actors in the system. However, more threating is the assumption that the propagation of invalid information is done with malicious intent. Here, one or many colluding actors might be spreading contradicting information to compromise the integrity of a system. Lamport uses an analogy called “The Byzanzine Generals Problem” to illustrate the scenario more clearly (Lamport, Shostak and Pease, The Byzantine Generals Problem 1982). He proposes the following scenario: The byzantine army is planning a siege on the enemy’s city. The attacking forces are split up in multiple divisions that have set up camp outside the city. Each division is led by one general that can send a messenger to other generals to communicate. After observing the enemy for some time, the generals must now agree on one course of action: either they all attack or they all retreat. The problem arises when some of the generals are traitors trying to prevent the others from agreeing on the next move. Lamport suggests that the loyal generals must follow an algorithm with a communication protocol that can guarantee that traitors cannot manipulate the decision-making process. He concludes that at least two thirds of the generals need to be honest to deny potential traitors to influence the decision. A consensus algorithm would therefore be considered byzantine failure tolerant if it can withstand faulty members in its system.

8.3. State machine Replication

A state machine is a system that is described by its distinct states and state transitions. Each state consists of a set of state variables and state transitions.

Through external input a state transition can be triggered, modifying the state variables, and optionally generating a form of output (Schneider 1990). Therefore, a state machine is deterministic as it can only be in a discrete set of states. That means, some state machines A and B that started in the same base configuration and then received the same set of inputs will end up in the same state with identical values for their state variables.

Many, if not all applications, can be broken down into discrete states and variables, allowing them to be represented as state machines (Schneider 1990). Replication can be utilized to make such an application and its underlying state machine fault tolerant. Replicating a state machine refers to the reproduction of the machine's current state including state variables on another state machine. When a system consists of multiple replicas of a state machine it allows for one or few of them to malfunction while the system is still operating as intended. The state is now maintained across multiple components of the system. To guarantee consistency of this state and its changes over time a protocol must be introduced that handles the communication between the different system components. This is where a consensus algorithm is utilized to ensure that each component receives the latest changes and all state machines across the system are kept in synchronization. Usually, when implementing replicated state machines, it is common to do so by using them in conjunction with a replicated log. A log is a structure that holds all inputs applied to the system in a sequence. Inputs are any external commands that invoke a change in the state machine. These inputs are always applied to the state machine exactly in the order how they were added to the log. Therefore, if the log is replicated correctly across the system, then all state machines are replicated correctly as well. This results out of the rule mentioned earlier, that the same consecutive inputs lead to the same outputs on two identical state machines.

8.4. Application

There are a great number of different consensus algorithms being used in distributed systems today. Most implementations are either based on Paxos or some deviation of it (Ongaro and Ousterhout 2014). Over the years a lot of adjustments have been made to original algorithm to make it faster, safer or to add some additional properties to it to fit specific use cases. An example of that is the Byzantine Paxos algorithm which extends the Paxos algorithm to be tolerant to a byzantine failure (Castro and Liskov 1999). A further extension of that is the fast byzantine algorithm that, as the name suggests, tries to reduce communication delay (Martin and Alvisi 2006).

Google uses Paxos for their lock service called Chubby (Burrows 2006). Chubby is used in distributed systems to handle synchronization and leader election amongst a set of servers. In particular, the Google File System is utilizing Chubby to provide fault tolerance while offering high performance to a great number of clients. The distributed file system is used internally at Google in data generation and processing for their services (Ghemawat, Gobioff and Leung 2003).

Especially in recent years the popularity of blockchain and cryptocurrencies has sparked a lot of public interest in distributed consensus. At the very core of these distributed peer-to-peer systems are consensus algorithms. The key difference between a lot of the systems mentioned previously and those running on a blockchain such as cryptocurrencies or other distributed services is often their availability to the public. A completely permissionless distributed system such as a public peer-to-peer network has no barrier of entry. Therefore, even byzantine fault tolerant consensus algorithms will not be able to protect the network from a sybil attack (Douceur 2002). A sybil attack is the intrusion of a network with multiple forged identities, allowing the attacker to take over the control of a peer-to-peer network that relies on multiple unaffiliated participants to maintain its integrity. As mentioned previously, for a consensus algorithm to be byzantine fault tolerant there always needs to be a certain majority of honest parties. If that threshold is broken there can be no guarantee that the system is

still operating as intended. Since there is no way to verify the identity of a participant in a completely permissionless distributed system there is also no mechanism to prevent the network from being overtaken by one or few colluding individuals that are wrongfully representing themselves as a multitude of entities inside the network. To counter the issue, permissionless consensus algorithms have been developed to allow for public peer-to-peer networks to be able to operate under similar conditions as private networks would.

9. The Raft Consensus Algorithm

Raft is a consensus algorithm that was developed with understandability in mind, as the title of the Raft paper reads: "In Search of an Understandable Consensus Algorithm" (Ongaro and Ousterhout 2014). Ongaro himself claims that Raft is "more understandable than Paxos and also provides a better foundation for building practical systems" (Ongaro and Ousterhout 2014). The difference between Raft and Paxos is in its structure while the goal of Raft is to still yield the same results and offer equivalent efficiency. To prove their claim of better understandability Ongaro and Ousterhout conducted an experiment amongst undergraduate and graduate students at Stanford University and U.C. Berkeley (Ongaro and Ousterhout 2014). In this experiment there were two groups of students each attending lectures about both Raft and Paxos. The Paxos lecture contained specifically the material necessary to create a replicated state machine equivalent to that of Raft. Now, one group attended the Raft lecture and then the Paxos lecture after while the other group did the same just in opposite order. According to Ongaro, this accounts for the different performance and gained experience of each participant from the first part of the study. To evaluate how well each lecture was understood, the participants were asked to take a test after each of them. Ongaro reports that the results of the concluded study reveal participants scored higher on the Raft test than they did on the Paxos one. The mean scores on the Raft and Paxos tests were 25.7 and 20.8, respectively. Therefore, participants performed better on the Raft test by an average of 4.9 points (Ongaro and Ousterhout 2014). In addition, participants were questioned

about which algorithm would be easier to implement and explain. According to Ongaro, an over-whelming majority considered Raft to be easier in both cases. However, he does add that “these self-reported feelings may be less reliable than participants’ quiz scores, and participants may have been biased by knowledge of our hypothesis that Raft is easier to understand.” (Ongaro and Ousterhout 2014). In the end, the intention of Raft can be summarized as following: to challenge the dominance of Paxos in the field of consensus algorithms and provide a more approachable and practical algorithm in the face of a growing demand for reliable distributed system supported by consensus.

As the Raft consensus algorithm is of central importance to this paper and the following section, the algorithm with its components and procedures will now be explained in detail.

9.1. The Algorithm

In a previous section the concept of replicated state machines has already been explained and how a consensus algorithm is utilized to achieve the replication of a machine’s state across an entire system of machines. This section will describe how exactly this replication is possible, specifically with the Raft consensus algorithm. As mentioned before, a replicated state machine is generally working together with a replicated log. Replicating the log means replicating the state machine, since it applies inputs exactly how they were added to the log. To keep said replicated log synchronized is the responsibility of the consensus algorithm.

A system implementing Raft, Ongaro refers to it as a Raft cluster (Ongaro and Ousterhout 2014), consists of a set number of servers. Any server can either be a leader, follower, or candidate. So, there are a total of three discrete states for a server, while most times there will only be one server assigned as the leader. This leader will be handling requests from clients and in case a follower should get a client request it will be redirected to the current leader. Each elected leader gets assigned a term, that is an integer that is increased with each election. At

the beginning of a term an election will be held, where one or multiple servers that were previously assigned the follower state become candidates and vouch to become the next cluster leader. It is guaranteed that at any given point there is only one active leader. The term functions as a way of ordering events, by comparing terms, outdated information can be detected such as a leader that is in fact no longer active. Each server stores the current term and updates it if it receives a message from another server with a higher term. Generally, each message that is sent from one server to another is prefixed with the sender's current term. When a leader or candidate receive a message with a higher term they immediately resign and enter the follower state. If any server receives a message with a lower term attached, it will consider the information to be obsolete and therefore ignore it (Ongaro and Ousterhout 2014).

9.1.1. Leader Election

Raft is a leader-based consensus algorithm, so to reach consensus there must first be an elected leader. The leader's job is to manage the replication of logs. It is doing so by accepting inputs from external clients, replicating said inputs to other servers and eventually notifying them when it is safe to apply the inputs to the underlying state machine. Initially all servers start in the follower state and remain in that state as long as they receive valid messages from a leader. A leader notifies all servers periodically in form of heartbeat messages to maintain its authority over the cluster. Each server has an individual election timeout, that is a set time chosen randomly from within a system wide election timeout range. Should a server not receive any messages from a leader within said election timeout it will initiate a new election as it has to assume that the current leader is no longer available or there is no leader to begin with (Ongaro and Ousterhout 2014).

When initiating an election, a follower increases its term by one and becomes a candidate. In this state, the candidate sends a message to all other servers asking them to vote for its leadership. If the vote is unanimous the candidate will go on to become the leader, now sending heartbeat messages to all others to prevent

any new elections from starting. However, should the candidate receive a message from another server with a term equal to or higher than its own then said candidate will step down and return into the follower state. In some cases, especially when multiple candidates happen to initiate elections around the same time, a split vote might occur and now there can be no majority for a single candidate. This means reelections need to be held until a new leader is successfully chosen (Ongaro and Ousterhout 2014).

9.1.2. Log replication

A leader, once elected, will handle all external requests from clients. A client sends commands to the leader which are then used as input for the replicated state machine. The leader assigns an index to this new input and then sends it to all other servers. The input is considered committed once a majority of servers have confirmed its reception. Only when that majority threshold is met will the leader apply the new input to its local log permanently. With each input committed by the leader its commit index is also increased to match the index of the committed input. Every heartbeat message from the leader contains said commit index along with the current term so that other servers can compare their commit index to that of the leader. If there is a difference between the two commit indices, then the server knows that there are new inputs that are safe to be committed. An input is only applied to the underlying state machine once said input is also committed on the log (Ongaro and Ousterhout 2014).

10. The Raft Consensus Algorithm in Multiplayer Games

As pointed out in this paper thus far, the concept of protecting distributed systems against potential failures and preserving availability and integrity is an already well-established field of research and development. Since there are many parallels between the issues of those distributed systems and peer-to-peer multiplayer games the feasibility of extending the application of consensus algorithms to real-time peer-to-peer multiplayer games will be the focus of this section.

First and foremost, a multiplayer game is a shared world in which the players expect to interact with each other and the environment. To keep the illusion of such a shared game world all changes to it must be communicated to all players. This requirement of a multiplayer game sounds quite similar to the concept of fault tolerance through replication described earlier. A distributed system aims to replicate all information in order to prevent losing it in case a part of the system experiences a failure. Similarly, in a peer-to-peer multiplayer game the state of the game world should not be lost just because one individual player decides to quite the game. In addition to mitigating loss of information the synchronization of the game state is essential to the very foundation of the multiplayer experience: Interaction. For an interaction between players to make sense the context in which the interaction happens needs to be the same for all participants. Therefore, the synchronization of information in a multiplayer game is absolutely necessary.

10.1. Multiplayer Game as Replicated State Machine

As mentioned before, most if not all applications can be modeled as some form of state machine. A game is not any different in that aspect. Certainly, a game

can have deep rooted systems with players, non-player characters and many other factors involved, but ultimately all these sub-components of the system are just state machines in and of themselves. So, when abstracting a game to only its variables and different states it can very well be represented as a state machine. Given that games are essentially just very complex state machines, a consensus algorithm could be utilized to synchronize the game state between different machines in a peer-to-peer network. Here, the input from players and the resulting actions are often the driving force behind changes in the game world. So, when applied in the context of multiplayer games, all a consensus algorithm would need to do is guarantee that all machines in the peer-to-peer network are executing the different player inputs in the right order and the synchronization of the game world then follows out of the deterministic nature of the state machine. Until this point, using consensus algorithms in peer-to-peer multiplayer games might offer an interesting approach to the synchronization of the game state, but that in and of itself has not necessarily been a pressing issue in the field.

A lot more troublesome is the issue of cheating in peer-to-peer multiplayer games as pointed out before. In less protected peer-to-peer games a cheating player can simply send false information to other players, and they will accept it as being true without any form of validation. The interesting approach to prevent cheating in peer-to-peer games comes from the idea behind byzantine fault tolerance. When comparing the role of a traitor in "the Byzantine Generals Problem" (Lamport, Shostak and Pease, The Byzantine Generals Problem 1982) to that of a cheater in peer-to-peer multiplayer games there is a lot of similarities in their methods of manipulation and the damaging results thereof. So, the natural question arises, whether it is possible to prevent cheating in a peer-to-peer multiplayer game by implementing it as a replicated state machine and letting a byzantine fault tolerant consensus algorithm handle all communications between peers.

10.2. Byzantine Failure in Raft

Consensus algorithms can be very complex and difficult to grasp, as Ongaro's Raft paper suggests (Ongaro and Ousterhout 2014) and hence the Raft consensus algorithm was developed. Because of that focus on simplicity, it felt natural to choose Raft as a starting point into the research of how consensus algorithms could be used for cheat prevention in peer-to-peer multiplayer games. Unfortunately, there is one distinct problem with using Raft in such a context and that is the algorithm's vulnerability to a byzantine failure. Since Raft is a crash failure tolerant and not a byzantine fault tolerant protocol the proposed correctness of the algorithm is no longer guaranteed once it is operating with faulty components in the system (Tan, Hu and Wang 2019). Tan, Hu, and Wang formulate a list of weaknesses that are exposed in Raft under byzantine practice. They state that Raft is a protocol with strong leadership, and followers simply accept any messages from the leader without any validation. If a byzantine member were to be elected as leader of the cluster it could tamper with the replicated log by wrongfully appending new entries or reordering the entries when another member requires an update due to previous inconsistencies (Tan, Hu and Wang 2019). In addition, the log replication process is further at risk due to the leader's ability to not notify certain followers about the confirmation of logs, which in turn would lead to a consensus failure and prevent the cluster from making progress (Tan, Hu and Wang 2019). According to Tan, Hu and Wang the leader election process is another large weakness in the protocol. Any follower can initiate a new election and the protocol will favor the candidate with the highest term and committed index as it suggests that this candidate has committed all or at least the most logs out of all members in the cluster. The problem arises when a candidate pretends to have a higher term and commit index than it really has, which will lead to said candidate to be almost guaranteed to win the election (Tan, Hu and Wang 2019). This is essentially a free passage for any faulty member of the cluster to claim its position as a leader and then abuse that power without any way for the rest of the cluster to protect themselves against it. Furthermore, a malicious follower could simply abuse the election mechanism by continuously initiating new elections even though the heartbeat

timeout was never reached. This would lead to a complete halt of the entire cluster since all members would be busy constantly voting for new leaders. This attack starves the system and renders it useless, as Raft is a leader-based consensus algorithm that cannot make any further progress for as long as there is not active leader.

10.3. Validation-Based Byzantine Failure Tolerant Raft

Tan, Hu and Wang not only pointed out the weaknesses of Raft in face of a byzantine failure, but also proposed a solution to the problems they identified: Validation-based byzantine failure tolerant Raft, an algorithm based on the principles of Raft that can withstand a total of *n/3* faulty members in a system with *n* total members (Tan, Hu and Wang 2019). They add two distinct features to Raft to guarantee byzantine failure tolerance. First, the communication between members is now digitally signed through cryptographic techniques to ensure the content of messages cannot be tampered with. In addition, log entries are now stored in a nested hash, a structure inspired by the blockchain, which requires each log entry to contain information about its preceding log. Linking the log entries together in such a nested structure makes it almost impossible to alter them after the fact (Tan, Hu and Wang 2019). In the following the changes Tan, Hu and Wang proposed will be explained in more detail.

Each member of the cluster is now adding a digital signature to their messages (Tan, Hu and Wang 2019). A digital signature is an authentication mechanism that lets the sender and content of a message be verified by a recipient through a unique code attached to the message (Kaur and Kaur 2012). According to Kaur and Kaur, this code or signature is made up of the encrypted hash of the message's content. The encryption is done with some asymmetric encryption, which is a form of encryption that uses a pair of keys to encrypt and decrypt messages (Kaur and Kaur 2012). The public key is used to decrypt a message that was encrypted with the private key. As the names suggest, the private key

is kept secret, while the public key is shared with potential recipients. Upon receiving a message, the signature is decrypted using the public key and following that the hash contained in the signature is compared to the hash of the plaintext in the message. If the two hashes match up, then the message is from the correct sender and was also not altered after the fact, on the other hand, should the hashes to not match it would suggest that the message has been tampered with.

Additionally, Tan, Hu and Wang added a log commit confirmation that is sent between the followers to prevent a potential byzantine leader to manipulate the log replication process. By involving all members into the confirmation process a faulty leader cannot forge false commits and push them to followers that will blindly accept them as true. Each follower waits for at least two thirds of all members to confirm a log before it is committed to the local log (Tan, Hu and Wang 2019).

Finally, the leader election process has been extended to prevent faulty members from winning elections (Tan, Hu and Wang 2019). First and foremost, a candidate does not receive votes anymore without being eligible to become a leader. The verification is done by each follower that is asked to vote for a specific candidate. The candidate attaches a batch of committed logs to the message when requesting a vote so that the follower can validate the logs and therefore prove the authenticity of the candidacy.
Tan, Hu and Wang ran a set of simulations to evaluate the performance of the Validation-based byzantine failure tolerant Raft algorithm in comparison to Raft. Their result showed that vbbft Raft “shares similar performance and scalability with Raft” (Tan, Hu and Wang 2019). Of course, there is some overhead that comes with adding additional messages for voting and commit confirmation, but apparently it does not seem to impact the performance drastically. There are many different hash algorithms with different key sizes, therefore also different hashing speeds (Kaur and Kaur 2012). The choice of the hash algorithm for the digital signatures can then lead to different results regarding performance.

10.4. Practicality of Raft as Cheat Prevention in Multiplayer Games

This section aims to answer the question whether Raft is an effective method of cheat prevention in peer-to-peer multiplayer games, considering the findings outlined in this paper. By doing so, the latency that the Raft protocol would add to a peer-to-peer multiplayer game needs to be examined to evaluate how practical this approach is. As a part of this paper, a simulation has been carried out to collect a benchmark for this key metric that is an essential bottleneck in multiplayer games. The simulation was recorded on a slightly modified implementation of the Raft algorithm that is a part of the DotNext Library (DotNext 2021). The project is written in C# and was built running on the .NET 5 framework. Due to limited resources in both time and computing power, the metrics extracted from the simulation will not be suitable when used as a precise measurement of the performance of Raft by any means. The sample rate was certainly too low for the data to be generalized and the system that was running the simulation could have also been a bottleneck[1]. Still, the data shows a general order of magnitude that gives an idea about what level of interactivity is still possible when running a peer-to-peer multiplayer game within a Raft cluster. The recorded data will be presented in Figure 3 and Figure 4, following that the results will be discussed and put into the context of multiplayer games.

[1] The simulation was carried out on a system running an Intel® Core ™ i7-10700 CPU 2.9GHz and 32 GB RAM, while the latency was simulated using clumsy (Jagt 2021). For each data set of the Time Per Commit simulation 100 commits were recorded and the average time was calculated. Same applies to the Time Per Election simulation except only 10 elections were recorded.

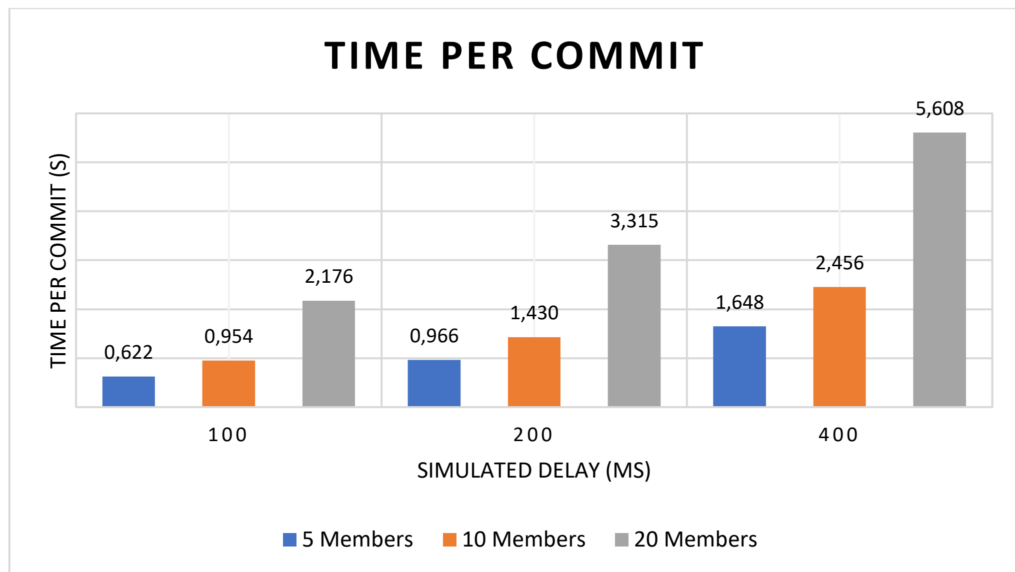

Figure 3. Time per Commit in a Raft Cluster with different number of members and varying simulated latency.

The time per commit represents the total time in seconds that it took for the leader of the Raft cluster to broadcast a newly added log entry to all members and wait for the responses of a majority of them (Figure 3). Once the quorum for that log is reached it is considered committed. The time it takes for a commit to be fully replicated in the entire cluster grows rapidly with the total number of members, since the leader needs to wait for the confirmation from all other members. The data sets are too small to make predictions about how the metric would develop in a cluster with even more members. However, the results from the current simulations can already be used a general benchmark for clusters with smaller size. The correlation between the time it takes for a log to be committed and the latency in the cluster is understandable as each time the leader tries to communicate with other members a delay is added.

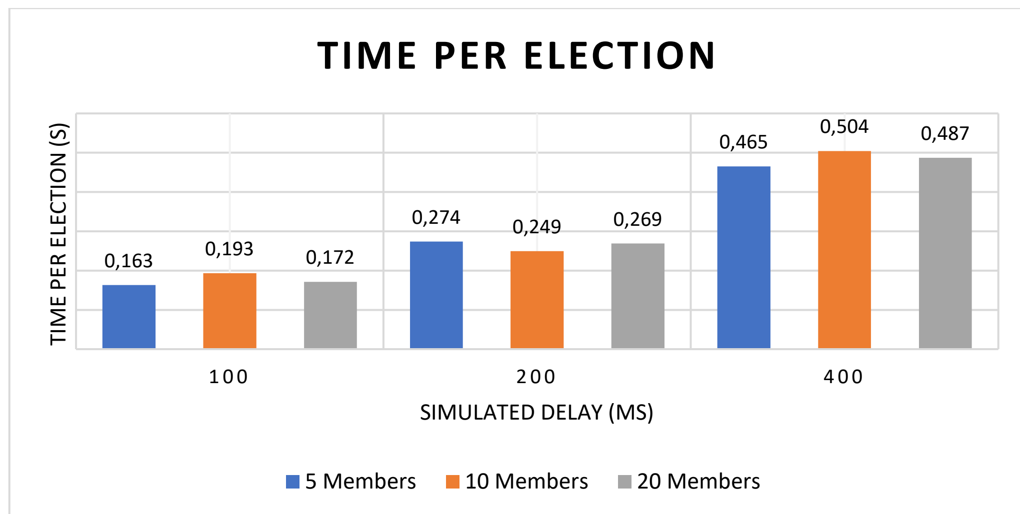

Figure 4. Time per Election in a Raft Cluster with different number of members and varying simulated latency.

The time per election is the total time it took for the members of the cluster to elect a new leader after the previous one resigned (Figure 4). When looking at the data gathered through the simulations it suggests that the leader election does take longer in clusters with higher latency between members. However, based on the data collected, the total number of members in the cluster seem to have no impact on the time it takes for the election of a new leader. Again, the data set is too small to generally suggest there is no correlation between the number of members in a cluster and the election time. The connection between the leader election and latency in the cluster makes sense for apparent reasons, as all communication in the cluster is delayed by it.

As seen in Figure 3 and Figure 4, Raft operates in magnitudes of hundreds or even thousands of milliseconds when committing logs and electing leaders. Usually, when referring to latency in multiplayer games the magnitude is in the tens or at most hundreds of milliseconds (Claypool and Claypool 2006). In a previous section the correlation between player performance and latency in multiplayer games has already been explored. The question if Raft, as it is, can be used as a protective measure against cheating has already been answered. As pointed out, Raft lacks critical safety properties when there are one or more byzantine members in the network. In the context of peer-to-peer multiplayer games a byzantine member is synonymous to a cheating player. Therefore, Raft

does not protect a peer-to-peer multiplayer game from any form of cheating. However, the previous section introduced vbbft Raft. In contrast to Raft, the algorithm claims to be byzantine failure tolerant under the condition that at least two thirds of the players are honest and are not cheating. Since the vbbft Raft is still recent, it was difficult finding an implementation to use for simulations and tests. Fortunately, Tan, Hu and Wang conducted benchmarks of vbbft themselves to compare its performance to that of Raft. Their conclusion was that vbbft Raft and Raft are analogous in terms of performance (Tan, Hu and Wang 2019). In accordance with their findings, the results from the simulations with Raft conducted in this paper could therefore also be considered a broad guideline for the results that the same simulations with vbbft Raft would yield. Ultimately, vbbft Raft adds complexity to the protocol so one could assume that it will not perform better than Raft if anything it would perform worse. So, for this purpose the conducted Raft simulations create a performance baseline for any Raft based algorithms that are building on top of it, while of course if an adapted algorithm aims to improve the performance this assumption might not hold true anymore.

To evaluate the practicality of Raft as cheat prevention in peer-to-peer multiplayer games the latency data from the simulations must be compared to those latency thresholds defined for multiplayer games earlier. After all, a game must remain interactable, so if a cheat prevention technique is interfering with the game in such a way that the base level of interactivity cannot be guaranteed then the technique might be just as disruptive as the cheating in the first place. Claypool and Claypool visualized their findings in a table that displays the beginning of the threshold of player tolerance towards latency (Table 1).

Model	Perspective	Example Genres	Sensitivity	Thresholds
Avatar	First-person	FPS, Racing	High	100 msec
	Third-person	Sports, RPG	Medium	500 msec
Omnipresent	Varies	RTS, Sim	Low	1,000 msec

Table 1. Latency threshold in multiplayer games

As seen in Table 1 the thresholds for the different game genres are starting well below one second as there is only one exception for games with low sensitivity

such as real-time strategy games and simulation games (Claypool and Claypool 2006). Realistically, this already lets us ignore any high sensitivity games, as the threshold is just too small for Raft to be deployed as a possible anti-cheat technique. Medium sensitivity games might work, but only under decent network conditions between the different players and a low number of players to even begin with. Ultimately, the only games where Raft might be a possible solution for cheat prevention are those with low sensitivity. As Figure 2 shows, the performance in low sensitivity games is moving linear to the latency (Claypool and Claypool 2006), therefore even delays up to *2000ms* might still result in enjoyable experiences for players. The performance in high and medium sensitivity games simply drops off to fast as latency increases.

11. Conclusion

The goal of this paper was to evaluate to what degree the Raft consensus algorithm can be used to prevent cheating in peer-to-peer multiplayer games. Unfortunately, during the research it was quickly apparent, that Raft does not meet the requirements to be applied in that context. However, this realization made it clear that it is not necessary for Raft to be able to prevent cheats and to evaluate exactly which specific cheats it could prevent. Of course, those findings would have been more practical and answered the question directly, but that might very well be work for the future. However, this paper does answer the question if it even makes sense to continue research towards using consensus algorithms such as Raft as a protective measure against cheating in peer-to-peer multiplayer games. From a players’ perspective, it would make no sense to impose an anti-cheat measurement onto a game, that might be successful in preventing cheats, but by doing so renders the game unplayable. Therefore, this paper has shown that, under real-world network conditions the implementation of Raft in most games is impractical due to the delay that is added to the communication between players. Although there is potential, maybe not for all peer-to-peer multiplayer games, but at least for those with low sensitivity player actions such as real-time strategy and simulation games. Here, further research

must be conducted with a byzantine fault tolerant algorithm such as vbbft Raft to test exactly which kind of cheats can be prevented and precisely how much network delay is added to the game.

12. References

Baughman, Nathaniel, Marc Liberatore, and Brian Levine. 2007. "Cheat-Proof Playout for Centralized and Peer-to-Peer Gaming." *IEEE/ACM Transactions on Networking.* IEEE. 1-13.

Burrows, Mike. 2006. "The Chubby lock service for loosely-coupled distributed systems." *Operating Systems Design and Implementation.* Seattle: USENIX Association. 335–350.

Castro, Miguel, and Barbara Liskov. 1999. "Practical Byzantine Fault Tolerance." *Proceedings of the Third Symposium on Operating Systems Design and Implementation* 173–186.

Claypool, Mark, and Kajal Claypool. 2006. "Latency and player actions in online games." *Communications of the ACM.* New York: Association for Computing Machinery. 40-45.

CoinMarketCap. 2021. *CoinMarketCap.* Accessed May 31, 2021. https://coinmarketcap.com/charts/.

DeLap, Margaret, Björn Knutsson, Honghui Lu, Oleg Sokolsky, Usa Sammapun, Insup Lee, and Christos Tsarouchis. 2004. "Is Runtime Verification Applicable to Cheat Detection?" *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games.* New York: Association for Computing Machinery. 134-138.

DotNext. 2021. *DotNext GitHub.* June 11. Accessed June 29, 2021. https://github.com/dotnet/dotNext.

Douceur, John R. 2002. "The Sybil Attack." *Peer-to-Peer Systems. IPTPS 2002. Lecture Notes in Computer Science.* Springer. 251–260.

ESA. 2020. *2020 Essential Facts About the Video Game Industry.* July. Accessed June 30, 2021. https://www.theesa.com/resource/2020-essential-facts/.

Foo, Chekyang, and Elina Koivisto. 2004. "Defining grief play in MMORPGs: player and developer perceptions." *ACE '04: Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology.* New York: Association for Computing Machinery. 245-250.

Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. 2003. "The Google File System." *Proceedings of the nineteenth ACM symposium on Operating systems principles.* New York: Association for Computing Machinery. 29-43.

Jagt. 2021. *Clumsy.* Accessed June 30, 2021. https://jagt.github.io/clumsy/index.html.

Johnson, Barry. 1984. "Fault-Tolerant Microprocessor-Based Systems." *IEEE Micro* 6–21.

Kabus, Patric, Wesley Terpstra, Mariano Cilia, and Alejandro Buchmann. 2005. "Addressing Cheating in Distributed MMOGs." *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games.* New York: Association for Computing Machinery. 1-6.

Kaur, Ravneet, and Amandeep Kaur. 2012. "Digital Signature." *2012 International Conference on Computing Sciences.* Phagwara: IEEE. 295-301.

Lamport, Leslie. 1998. "The Part-Time Parliament." *ACM Transactions on Computer Systems* 133-169.

Lamport, Leslie, Robert Shostak, and Marshall Pease. 1982. "The Byzantine Generals Problem." *ACM Transactions on Programming Languages and Systems* 382-401.

Lan, Xiao, YiChun Zhang, and Pin Xu. 2009. "An Overview on Game Cheating and Its Countermeasures." *Proceedings of the Second Symposium International Computer Science and Computational Technology.* Huangshan City. 195-200.

Martin, Jean-Philippe, and Lorenzo Alvisi. 2006. "Fast Byzantine Consensus." *IEEE Transactions on Dependable and Secure Computing* 202–215.

Mulligan, Jessica, and Bridgette Patrovsky. 2003. *Developing Online Games: An Insider's Guide.* New Riders Games.

Nakamoto, Satoshi. 2008. *Bitcoin: A Peer-to-Peer Electronic Cash System.* October 31. Accessed June 30, 2021. https://bitcoin.org/bitcoin.pdf.

Ongaro, Diego, and John Ousterhout. 2014. "In Search of an Understandable Consensus Algorithm." *Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.* Philadelphia: USENIX. 305-319.

Schneider, Fred. 1990. "Implementing fault-tolerant services using the state machine approach: a tutorial." *ACM Computing Surveys* 299-319.

SuperData. 2020. *2019 Year in Review.* SuperData, A Nielsen Company.

Tan, Dezhi, Jianguo Hu, and Jun Wang. 2019. "VBBFT-Raft: An Understandable Blockchain Consensus Protocol with High Performance." *2019 IEEE 7th International Conference on Computer Science and Network Technology (ICCSNT).* Dalian: IEEE. 111-115.

Webb, Steven, and Sieteng Soh. 2007. "Cheating in networked computer games – A review." *DIMEA '07: Proceedings of the 2nd international conference on Digital interactive media in entertainment and arts.* New York: Association for Computing Machinery. 105-112.

Yan, Jeff, and Brian Randell. 2009. "An Investigation of Cheating in Online Games." *IEEE Security & Privacy*, May-June: 37-44.