Department of Computing

Referee-Based Architectures for Massively Multiplayer Online Games

Steven Daniel Webb

This thesis is presented for the degree of

Doctor of Philosophy

of

Curtin University of Technology

November 2010

To the best of my knowledge and belief this thesis contains no material previously published by any other person except where due acknowledgement has been made. This thesis contains no material which has been accepted for the award of any other degree or diploma in any university.

04-11-2010

Steven Daniel Webb

Abstract

Network computer games are played amongst players on different hosts across the Internet. Massively Multiplayer Online Games (MMOG) are network games in which thousands of players participate simultaneously in each instance of the virtual world. Current commercial MMOG use a Client/Server (C/S) architecture in which the server simulates and validates the game, and notifies players about the current game state. While C/S is very popular, it has several limitations: (i) C/S has poor scalability as the server is a bandwidth and processing bottleneck; (ii) all updates must be routed through the server, reducing responsiveness; (iii) players with lower client-to-server delay than their opponents have an unfair advantage as they can respond to game events faster; and (iv) the server is a single point of failure.

The Mirrored Server (MS) architecture uses multiple mirrored servers connected via a private network. MS achieves better scalability, responsiveness, fairness, and reliability than C/S; however, as updates are still routed through the mirrored servers the problems are not eliminated. P2P network game architectures allow players to exchange updates directly, maximising scalability, responsiveness, and fairness, while removing the single point of failure. However, P2P games are vulnerable to cheating. Several P2P architectures have been proposed to detect and/or prevent game cheating. Nevertheless, they only address a subset of cheating methods. Further, these solutions require costly distributed validation algorithms that increase game delay and bandwidth, and prevent players with high latency from participating.

In this thesis we propose a new cheat classification that reflects the levels in which the cheats occur: game, application, protocol, or infrastructure. We also propose three network game architectures: the Referee Anti-Cheat Scheme (RACS), the Mirrored Referee Anti-Cheat Scheme (MRACS), and the Distributed Referee Anti-Cheat Scheme (DRACS); which maximise game scalability, responsiveness, and fairness, while maintaining cheat detection/prevention equal to that in C/S. Each proposed architecture utilises one or more trusted referees to validate the game simulation - similar to the server in C/S - while allowing players to exchange updates directly - similar to peers in P2P.

RACS is a hybrid C/S and P2P architecture that improves C/S by using a referee in the server. RACS allows honest players to exchange updates directly between each other, with a copy sent to the referee for validation. By allowing P2P communication RACS has better

responsiveness and fairness than C/S. Further, as the referee is not required to forward updates it has better bandwidth and processing scalability. The RACS protocol could be applied to any existing C/S game. Compared to P2P protocols RACS has lower delay, and allows players with high delay to participate. Like in many P2P architectures, RACS divides time into rounds. We have proposed two efficient solutions to find the optimal round length such that the total system delay is minimised.

MRACS combines the RACS and MS architectures. A referee is used at each mirror to validate player updates, while allowing players to exchange updates directly. By using multiple mirrored referees the bandwidth required by each referee, and the player-to-mirror delays, are reduced; improving the scalability, responsiveness and fairness of RACS, while removing its single point of failure. Direct communication MRACS improves MS in terms of its responsiveness, fairness, and scalability. To maximise responsiveness, we have defined and solved the Client-to-Mirror Assignment (CMA) problem to assign clients to mirrors such that the total delay is minimised, and no mirror is overloaded. We have proposed two sets of efficient solutions: the optimal J-SA/L-SA and the faster heuristic J-Greedy/L-Greedy to solve CMA.

DRACS uses referees distributed to player hosts to minimise the publisher / developer infrastructure, and maximise responsiveness and/or fairness. To prevent colluding players cheating DRACS requires every update to be validated by multiple unaffiliated referees, providing cheat detection / prevention equal to that in C/S. We have formally defined the Referee Selection Problem (RSP) to select a set of referees from the untrusted peers such that responsiveness and/or fairness are maximised, while ensuring the probability of the majority of referees colluding is below a pre-defined threshold. We have proposed two efficient algorithms, SRS-1 and SRS-2, to solve the problem.

We have evaluated the performances of RACS, MRACS, and DRACS analytically and using simulations. We have shown analytically that RACS, MRACS and DRACS have cheat detection/prevention equivalent to that in C/S. Our analysis shows that RACS has better scalability and responsiveness than C/S; and that MRACS has better scalability and responsiveness than C/S, RACS, and MS. As there is currently no publicly available traces from MMOG we have constructed artificial and realistic inputs. We have used these inputs on all simulations in this thesis to show the benefits of our proposed architectures and algorithms.

Acknowledgements

This thesis would not have been possible without the support of many people. In particular, I am extremely grateful to my supervisor Dr Sieteng Soh for the support and encouragement he has given me throughout my PhD. He was a tremendous supervisor and a good friend. Without his help I doubt I would have reached half way. I would also like to express my thanks to Dr William Lau for his assistance at the start of my PhD, and Dr Jerry Trahan for his assistance towards the end.

This thesis would not have been possible without the support of my girlfriend Elissa, my parents, my brother Ben, my Granny, and all of my friends. Thank you for everything you add to my life. I would not be here without you.

Thank you to everyone in the Department of Computing for their support and encouragement. In particular, Dr Patrick Peursum, Dr Mike Robey, and Richard Palmer.

Finally, I would like to thank the Australian government for providing me with the Australian Postgraduate Award (APA) scholarship. I would have been unable to undertake a PhD without this financial support.

Contents

List of Figures

List of Tables

List of Algorithms

Published Work

This thesis is based upon several works that have been published over the course of the author's PhD, listed as follows in chronological order:

- Webb, S. D., W. Lau, and S. Soh (2006). NGS: An application layer network game simulator. In *Proc. Interactive Entertainment (IE)*, pp. 15-22. Permission was obtained to reuse this copyrighted work (see Appendix B).
- Webb, S. D., S. Soh, and W. Lau (2007). RACS: a referee anti-cheat scheme for P2P gaming. In *Proc. Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pp. 37-42.
- Webb, S. D., S. Soh, and W. Lau (2007). Enhanced mirrored servers for network games. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 117-122.
- Webb, S. D. and S. Soh (2007). Cheating in networked computer games - a review. In *Proc. Digital Interactive Media in Entertainment and Arts (DIMEA)*, pp. 105-112.
- Webb, S. D. and S. Soh (2007). Round length optimisation for P2P network gaming. In *Proc. Postgraduate Electrical Engineering and Computing Symposium (PEECS)*, pp. 23-28 (best presentation award).
- Webb, S. D. and S. Soh (2007 (Published 2008)). A survey on network game cheats and P2P solutions. *Australian Journal of Intelligent Information Processing Systems 9*(4), 34-43.
- Webb, S. D. and S. Soh (2008). Adaptive client to mirrored-server assignment for massively multiplayer online games. In *Proc. Multimedia Computing and Networking (MMCN)*, pp. 6818-17.
- Webb, S. D., S. Soh, and J. L. Trahan (2008). Secure referee selection for fair and responsive peer-to-peer gaming. In *Proc. Principles of Advanced and Distributed Simulation (PADS)*, pp. 63-71 (best paper nomination).
- Webb, S. D. and S. Soh (2009). Application performance metrics for evaluating delay estimation schemes. In *Proc. Asia-Pacific Conference on Communications (APCC)*, pp. 717-721.

- Webb, S. D., S. Soh, and J. L. Trahan (2009). Secure referee selection for fair and responsive peer-to-peer gaming. *SIMULATION: Transactions of The Society for Modelling and Simulation International 85*(9), 608-618. Permission was obtained to reuse this copyrighted work (see Appendix B).

Acronyms, Abbreviations, and Notation

Common Acronyms and Abbreviations

AoI	Area of Interest.
BS	Bucket Synchronisation.
C/S	Client/Server.
CDS	Cheat Detection System.
CMA	Client-to-Mirror Assignment.
DRACS	Distributed Referee Anti-Cheat Scheme.
E-Mirror	Egress Mirror.
FPS	First Person Shooters.
I-Mirror	Ingress Mirror.
MMOG	Massively Multiplayer Online Games.
MPP	Peer-to-peer message.
MPR	Peer-to-referee message.
MRACS	Mirrored Referee Anti-Cheat Scheme.
MRP	Referee-to-peer message.
MRR	Referee-to-referee message.
MS	Mirrored Server.
ODL	On Demand Loading.
P2P	Peer-to-Peer.
PP	Peer-Peer mode for RACS, MRACS, and DRACS.
PRP	Peer-Referee-Peer mode for RACS, MRACS, and DRACS.
RACS	Referee Anti-Cheat Scheme.
RTT	Round Trip Time.
TSS	Trailing State Synchronisation.
WoW	World of Warcraft.

Common Notation

τ	Round length.
r	Round number.
$d_{i,j}$	Delay from host i to host j.
R	The set of referees.
R_i	A referee with ID i.
M	The set of Mirrors.
M_i	A mirror with ID i.
m	The number of mirrors, |M|.
P	The set of players.
P_i	A player with ID i.
n	The number of players, |P|.
S_A	The authentication server.

Chapter 1

Introduction

Computer games have become a mainstream form of entertainment, making gaming a highly profitable market. A report sponsored by the *Entertainment Software Association* estimated the total sales in 2007 for the US computer game industry at $18.85 billion [123]. Network computer games are competitive and/or cooperative games played amongst multiple players on different hosts across a network, often the Internet. The popularity of network games has increased rapidly, made feasible by the increase in household penetration of broadband Internet access [14]. Most network games support several dozen simultaneous players in each instance of the virtual world. Massively Multiplayer Online Games (MMOG) differ from traditional network games as they present a single virtual world in which tens of thousands of players participate simultaneously [27]. Furthermore, these worlds are persistent; hence, the state of the world evolves even when the player is offline [27]. Therefore, in addition to addressing game responsiveness, fairness, consistency, and cheat-free requirements, MMOG must address scalability, persistency, and reliability [6, 43, 71]. Despite the additional complexity, MMOG are popular amongst game developers and publishers as they can be extremely profitable [137]. This thesis does not attempt to address all of these issues, focusing on scalability, responsiveness, fairness, and cheating.

In this thesis the scalability of an architecture is defined as its ability to support a large number of concurrent players, and tolerate a rapid increase in the number of concurrent players, without dramatically increasing the usage of centralised resources [71]. Scalability is a critical factor when designing the architecture for MMOG, as it must support tens of thousands of concurrent players. Further, as it is difficult to predict the popularity of a game at launch time, the architecture should tolerate an unexpected dramatic increase in players, without the need to provision significant additional expensive infrastructure.

The most common approach to improve scalability is sharding [24]. A shard is a complete and independent copy of the game world. The developer determines the maximum number of concurrent players per shard. By adding more shards the developer can accommodate more players; however, players in different shards cannot interact, thus sharding works against the concept of MMOG. Furthermore, it is frustrating and annoying for players when shards reach their limits as they must play on different shards with different people, destroying the social aspect and therefore its appeal.

As games are arguably one of the most delay sensitive Internet applications, maximising game responsiveness is critical to provide players with a satisfying experience [10, 43, 100]. Further, if a player has higher delay than his[1] opponents, he is at a significant disadvantage, and may become frustrated and quit the game. Finally, even if all players have equal delay, the game must be responsive to maintain player satisfaction. As many MMOG use a subscription business model (players pay a monthly subscription fee) maintaining responsiveness and fairness is critical to the business.

The delay requirement for a game is dependent on its genre. First Person Shooters (FPS) are the most delay sensitive of games, with users preferring a delay below 180*ms* [10], while other genres of games can tolerate higher delays, even exceeding 1000*ms* [61]. The importance of delay for player satisfaction was emphasised by the developer of EVE Online, CCP Games, when it changed service providers to reduce the server to player delay [114]. CCP Games is also considering giving players control about how game updates are routed from EVE Online's data centre; thus, increasing both responsiveness and fairness.

To be fair, a game must also prevent players from cheating. While the majority of players are honest [100], a small population of cheaters can destroy an online community, as honest players often quit if cheating continues unpunished. This is catastrophic for games using subscriptions to generate revenue [100]. The importance of cheat prevention to developers and publishers is evident from the increasing number of commercial anti-cheat products [7, 46, 50, 104, 134]. Cheat prevention is one of the main benefits of using a Client/Server (C/S) game architecture.

The vast majority of MMOG use a C/S architecture, in which the server is the game authority. To support a virtual world with tens of thousands of players, the server is often comprised of multiple hosts at one location, with the game simulation distributed between them. With only one centralised trusted server, keeping the game consistent, persistent, and cheat free in C/S is straightforward [27, 77, 108]. Unfortunately, C/S suffers from the following limitations:

- bandwidth scalability - the server's incoming and outgoing bandwidth are bottlenecks as the publisher must provision sufficient bandwidth at one location, which is an expensive recurring cost [27, 94, 100];

- processing scalability - the server's processing power is a bottleneck, as it must simulate the entire virtual world and perform Area of Interest (AoI) filtering for all players in real time[128];

- responsiveness - each player update is sent to the server, which simulates the update, and broadcasts the new game state to relevant players. Thus, the game delay for C/S

[1]Note, "he" should be read as "he or she" throughout this thesis.

is at least two times the Round Trip Time (RTT) [94];

- fairness - players with low client-to-server delay have an unfair advantage, as they experience better responsiveness than those situated further away [43]; and

- reliability - the server is a single point of failure for the system [16, 43].

Many network game architectures have been proposed to overcome the weaknesses of C/S. The Mirrored Server (MS) architecture [41–43], comprising multiple mirrored servers deployed at geographically different locations throughout the network, has better bandwidth scalability, responsiveness, fairness, and reliability than C/S. MS, however, requires a synchronisation mechanism to maintain consistency across mirrors, incurring additional processing overhead. Further, like in C/S, game delay in MS is not optimal as player updates are routed through the mirrors.

Many Peer-to-Peer (P2P) network game architectures have been proposed to address the C/S and MS limitations (*e.g.*, references [13, 17, 36, 64, 71, 76, 84]). P2P is scalable as the bandwidth and processing requirements are entirely handled by the peers; hence, there is no central bottleneck. Furthermore, P2P systems are resource growing; as the number of peers/players increases so does the overall bandwidth and processing power of the system. Finally, as updates can be sent directly between peers, P2P can potentially maximise responsiveness and fairness. Unfortunately, keeping the game consistent and cheat-free in P2P is significantly harder and more costly than in C/S and MS, as P2P does not have trusted servers/peers to store the authoritative game state and validate player updates [76]. Addressing cheating is a major concern for network games as it degrades the experience of the majority of players who are honest [100]. This is catastrophic for games using subscription models to generate revenue.

Several P2P protocols have been proposed to address cheating [13, 37, 64]. However, these protocols are vulnerable to the information exposure and invalid command cheats which are prevalent in MMOG, while introducing new forms of cheating not possible in C/S; see Section 2.2. In addition, these solutions require costly distributed validation algorithms that increase game delay and bandwidth. Finally, these protocols prevent players with high delay from participating.

In this thesis we propose using referees as a trusted third party to build scalable, responsive, fair, and cheat-free network game architectures. Our referee model was inspired by referees in traditional sporting events, who are responsible for ensuring the rules are followed, detecting / preventing cheating, but are not responsible for notifying players about the current state of play. Equivalently, in our proposed architectures, we have used trusted hosts running referees that are responsible for validating player updates, but are not required to notify players about the current game state unless cheating is detected. In contrast, the server in C/S must constantly notify all players about the current state of play, which consumes significant server bandwidth and processing power. On the other hand, naive P2P architectures do not validate player actions, and thus do not prevent cheating. This thesis proposes three network game architectures that apply the referee concept: the Referee Anti-Cheat Scheme (RACS), the Mirrored Referee Anti-Cheat Scheme (MRACS), and the Distributed Referee Anti-Cheat Scheme (DRACS). Conceptually, each of these three proposed architectures is a hybrid between C/S and P2P. For honest players, our architectures use P2P communication with players sending and receiving updates directly between each other, while the referee independently simulates and validates player updates. For cheaters, our architectures are equivalent to C/S in which players send updates to the referee, which simulates each update, and broadcasts the results to relevant players. Each architecture significantly improves C/S scalability, responsiveness, and fairness, while providing cheat prevention equal to that in C/S.

1.1 Aims and Approach

Our aim was to design three network game architectures that emphasise game scalability, responsiveness, and fairness , while providing cheat prevention equal to that in C/S. As requirements differ between games, a *"one size fits all"* approach will not be effective. Therefore, we decided on three specific aims, addressing different needs. In particular:

Aim 1 - To propose a network game architecture using a referee in the server to improve C/S scalability, responsiveness, and fairness: As C/S is the dominant network game architecture, and ubiquitous for MMOG, any improvement could potentially be used in many existing and future games. Therefore, Aim 1 was to create a hybrid C/S and P2P architecture with the security of C/S and the scalability of P2P; we call it the Referee Anti-Cheat Scheme (RACS). Placing a referee at the server maintains cheat prevention equal to C/S, but allows peers to exchange updates directly, increasing scalability, responsiveness, and fairness.

Aim 2 - To propose a network game architecture using multiple referees in mirrored servers to improve RACS and MS scalability, responsiveness, and fairness: The MS architecture provides better scalability, responsiveness, and fairness than C/S, provided the publisher can provision multiple mirrored servers connected via a private network. However, as updates are routed through the mirrors, like in C/S, the scalability, responsiveness, and fairness of MS are not optimal. Our Aim 2 was to place a referee at each mirror - mirrored referees - to gain the same scalability, responsiveness, and fairness benefits as in Aim 1, while preventing cheating. We call this architecture the Mirrored Referee Anti-Cheat Scheme (MRACS).

Aim 3 - To propose a network game architecture using referees in peers to improve P2P cheat detection/prevention: Aims 1 and 2 increase the scalability, responsiveness, and fairness of cheat-proof architectures. Aim 3 is the inverse in that we seek to increase the cheat-prevention properties of a scalable, responsive, and fair P2P architecture. In particular, for Aim 3, we propose the Distributed Referee Anti-Cheat Scheme (DRACS). This architecture probabilistically deploys referees to untrusted peers such that it satisfies a required degree of cheat detection / prevention, while maximising responsiveness and/or fairness.

1.2 Contributions

This thesis has the following seven contributions, presented in order of appearance.

1) A new cheat classification scheme. We have proposed a new classification covering all known cheats. Our classification is a modification and expansion of the scheme proposed by GauthierDickey *et al.* [64]. In addition, we have provided a review of all known theoretical and practical cheating techniques. Where applicable, real-world examples of these cheats are discussed. We use this classification in Chapters 3, 4, and 5 to evaluate the anti-cheat properties of our proposed architectures. Our classification has proved useful to other researchers investigating network games (*e.g.,* [66, 67, 96]). Further, by providing real world examples game developers can clearly link the different cheats to real-world threats.

2) The Referee Anti-Cheat Scheme (RACS). RACS is a hybrid C/S and P2P architecture that improves the bandwidth and processing scalability of C/S, and increases responsiveness and fairness, without increasing the opportunities for cheating. Our analysis and simulations in Chapter 3 show the reduction in bandwidth and increase in responsiveness of RACS over C/S. Any existing C/S architecture could potentially be adapted to use the RACS protocol.

3) The round length adjustment problem definition and its efficient solutions. In Chapter 3 we propose two efficient and accurate round length adjustment algorithms: *brute force* and *voting*. Both centralised algorithms are preferable for RACS than the distributed approach in [64]. Our simulations using real-world data traces show our algorithms effectiveness.

4) Constructed realistic inputs for game architecture evaluation. At present there are no publicly available traces from MMOG, making simulation and evaluation of new network game architectures difficult. Further, as C/S architectures do not measure the delay between clients, it is difficult to evaluate novel P2P architectures. To overcome these problems we generate realistic inputs that utilise real-world data traces from a popular *Counter-Strike* game server [54], the *hostip.info* IP address to geographic location mapping service

[68], the *Dimes project* Internet topology trace [121], the *PingER* project Internet performance measurements [93], and public information about *World of Warcraft* (*WoW*) [19]. Using these inputs we construct realistic simulations to evaluate our proposed architectures and algorithms.

5) The Mirrored Referee Anti-Cheat Scheme (MRACS). MRACS combines the RACS protocol with the MS architecture to improve the bandwidth scalability, responsiveness, and fairness of both architectures, and thus C/S. MRACS also improves the reliability of RACS and C/S by removing the single point of failure. By using the RACS protocol, MRACS provides security equal to that in C/S. Our theoretical analysis and simulations in Chapter 4 show the reduction in bandwidth, processing, and delay of MRACS over C/S, RACS, and MS.

6) The Client-to-Mirror Assignment (CMA) problem definition and and its efficient solutions. Chapter 4 introduces the CMA problem for minimising delay in heterogeneous mirrored systems with fixed resources. We propose an optimal solution, and a faster heuristic solution. Our discussion focuses on MS and MRACS; however, our solutions are applicable to all applications with mirrored resources and long-term connections (*e.g.,* Content Distribution Networks (CDN) [78] or video streaming sites such as *Hulu* [72]). Simulations using our realistic inputs show our CMA solutions significantly reduce delay.

7) The Distributed Referee Anti-Cheat Scheme (DRACS). DRACS combines the RACS protocol with the voting mechanism proposed by Kabus and Buchmann [76] to create a cheat resistant P2P network game architecture. For DRACS, we have defined the Referee Selection Problem (RSP) and propose two possible solutions - one maximising responsiveness and the other fairness. Both solutions maintain a developer defined level of security. While these algorithms are discussed in the context of RACS they are also applicable to other P2P network game architectures, *e.g.*, Region Controllers [76], NEO [64], SEA [36], FreeMMG [27], *etc.* Our simulations using real-world data traces show the effectiveness of both algorithms, and thus DRACS.

1.3 Thesis Organisation

The contents of each chapter in this thesis are as follows.

Chapter 2: Background and Literature Review

Chapter 2 includes background information and related work on three important areas of this research: network game properties, game cheating, and network game architectures (C/S, MS, and P2P).

Chapter 3: The Referee Anti-Cheat Scheme (RACS)

Chapter 3 proposes a new cheat classification that incorporates all known theoretical and practical cheats, presents RACS, formally defines the round length adjustment problem, and proposes two centralised algorithms for round length adjustment. Further, this chapter describes the construction of the artificial and realistic inputs used in all simulations throughout this thesis. It also includes analytical analysis and simulations to evaluate the bandwidth and delay improvements of RACS over C/S. Finally, the chapter evaluates our round length adjustment algorithms using simulation.

Chapter 4: The Mirrored Referee Anti-Cheat Scheme (MRACS)

Chapter 4 proposes MRACS, formally defines the CMA problem, and proposes two solutions for this problem. The chapter includes analytical analysis to compare the required bandwidth and processing requirements of RACS, MS, and MRACS. Finally, Chapter 4 includes simulations using the artificial and realistic inputs constructed in Chapter 3 to evaluate RACS, MS, and MRACS; and to evaluate the different CMA algorithms.

Chapter 5: The Distributed Referee Anti-Cheat Scheme (DRACS)

Chapter 5 proposes DRACS, formally defines the RSP, and proposes two solutions for this problem. Both solutions ensure the probability of colluding referees being below a developer specified level. We perform simulations using the artificial and realistic inputs constructed in Chapter 3 to evaluate both solutions.

Chapter 6: Conclusion

Chapter 6 summarises this thesis and discusses possible areas of future research.

Chapter 2

Background and Literature Review

This chapter is divided into three main sections: network game properties, game cheating, and network game architectures. Section 2.1 describes four common game genres, and reviews six properties that an MMOG architecture must address: scalability, responsiveness, fairness, consistency, persistence, and reliability [71]. While all six properties are important for MMOG, this thesis focuses on scalability, responsiveness, and fairness; however, consistency, persistence, and reliability are discussed where relevant. Section 2.2 describes our cheat model, describes all known theoretical and practical cheating techniques (including examples where applicable), and reviews previous classifications of cheating. Section 2.3 reviews previously proposed network game architectures, including C/S, MS, and several P2P architectures. Finally, Section 2.4 summarises this chapter. Note, the review of cheating in Section 2.2 was originally published in [139] and expanded in [141].

2.1 Network Game Properties

2.1.1 Network Game Genres

Due to their different game play, different game genres have different requirements for responsiveness and fairness. This section describes several popular genres of games.

An avatar is the virtual world representation of a player. Depending on the game genre, each player is represented by one or more avatars. In First Person Shooters (FPS) each player controls a single humanoid avatar in a 3D environment. The player's view is first person, and the goal is to collect weapons, and use them to shoot and kill other avatars. The vast majority of FPS are competitive, and success requires accurate aiming and quick reflexes (FPS are often called twitch games for this reason). Further, avatar movement is extremely fast and unpredictable [135]. Therefore, FPS generate many small packets at short intervals [54], and have very strict delay and fairness requirements. Thousands of independent copies of the game world, called instances, are hosted on servers all over the world; with each instance supporting several tens of players. While many servers run continuously, time is usually divided into independent rounds, with the game map cycled between each round. Example FPS include *Quake* [73], *Counter-Strike* [54], *Unreal Tournament* [14], *etc.*

Real Time Strategy Games (RTS) involve simulated armies battling each other. Each player is represented by an army, involving many avatars controlled using player guided Artificial Intelligence (AI). Similar to chess, the winner of an RTS game is the player with the best strategy. While higher responsiveness is beneficial, it is far less critical for RTS than FPS games, making fairness easier to achieve. Each instance of an RTS is typically limited to 10 or 20 players; however, smaller matches are common. While each instance is independent, RTS games often provide global player rankings according to wins and losses. Examples include *Age of Empires* [16] and *Warcraft* [122].

Role Playing Games (RPG) have a strong story and involve a player controlled avatar embarking on quests or missions with other players. The objective is for each player to improve his avatar's skills, abilities, levels, and equipment. Game play is often designed such that quick reflexes are not a requirement to be successful. Thus, the required responsiveness is comparable to RTS, and significantly lower than FPS. Further, as RPG are often cooperative, fairness is not as critical. Each RPG instance typically supports up to 10 players. Examples include *Diablo* [1] and *Fallout* [2].

Massively Multiplayer Online Games (MMOG) differ from traditional network games as they present a single universe in which thousands or tens of thousands of players participate simultaneously [84]. Further, the state of the game world and players' avatars progress gradually, lasting months or years. In the last five years the popularity of MMOG has increased dramatically; enabled by the explosive growth of the Internet and the availability of broadband connections for home users. While massively multiplayer FPS and RTS games have been created, Massively Multiplayer Role Playing Games (MMORPG) are the dominant form of MMOG. As MMORPG have similar game play to traditional RPG (but involving more players) they have comparable responsiveness requirements. However, while MMORPG have a strong focus on cooperation, they can also be very competitive, making fairness a higher priority than in RPG. Examples include *World of Warcraft* (*WoW*) [19] and *EverQuest* [85].

To ensure compatibility with the widest range of genres this thesis focuses on the scalability requirements of MMOG, and the responsiveness and fairness requirements of FPS. Thus, our proposed architectures are applicable to all genres of games.

2.1.2 Scalability

We define the scalability of an architecture as its ability to support a large concurrent player population, and tolerate a rapid increase in the concurrent player population, without dramatically increasing the usage of centralised resources. For a C/S architecture the server is a potential bottleneck, limiting scalability, as each additional client consumes more of the server's bandwidth and processing power [13, 43, 64, 67, 108]. To prevent a bottleneck occurring publishers must provision large amounts of hardware and bandwidth [13, 84, 101]; as the connected population fluctuates diurnally these resources are frequently underutilised [55, 83, 110, 130]. Further, as more resources cannot be deployed rapidly, publishers often over-provision to allow for a rapid increase in players [28]. The cost of provisioning sufficient resources makes MMOG development difficult and risky for small companies, and even some large developers struggle to provision sufficient resources [85, 132].

For games involving many players it is impractical to update each player about every change in game state. It is financially impractical as the server's bandwidth is an expensive recurring cost [100]; and it is technically impractical as players do not have sufficient bandwidth. To reduce the server to client traffic each player is only notified about relevant game state changes; this is called *Interest Management* [124]. A player's *Area of Interest* (AoI) is the portion of the game world they can perceive and influence; each player only receives notifications about state changes that occur within their AoI. By reducing the data sent to each client the server's bandwidth scalability is increased, and players with low bandwidth can participate.

For the server to simulate a virtual world involving thousands of simultaneous participants requires distributing the workload across multiple hosts; *i.e.*, the server is a cluster of cooperating hosts [23, 24, 33, 85]. A common approach to distribute the workload is to divide the virtual world into zones/regions of fixed [30, 58, 109, 154] or varying [92] sizes. Each host simulates one or more zones [30, 88, 128], and a load balancing algorithm distributes the workload amongst the cluster of hosts. However, as the number of player interactions increases the processing requirements increase faster than linearly [8, 109]; thus, a large congregation of players can potentially overload the server. To prevent this from occurring the game will often limit the maximum number of players in each region [109].

Load balancing across a cluster allows MMOG to support tens of thousands of players; however, to support hundreds of thousands or millions of players requires sharding the virtual world [109]. A shard is a complete and independent copy of the game world. By provisioning shards at different locations the required bandwidth is distributed throughout the network, reducing the bandwidth bottleneck and the client-to-server delay. To prevent a processing bottleneck the maximum number of concurrent players per shard is bounded. By adding more shards the developer can accommodate more players; however, as players in different shards cannot interact this works against the concept of MMOG. Further, it is frustrating and annoying for players when shards reach their limits as they must play on different shards with different people, destroying the social aspect of MMOG.

A prominent example of a sharded MMOG is *World of Warcraft* (*WoW*), which is arguably the most popular MMOG to date, with over 10 million players worldwide [110]. The WoW universe is sharded into many mutually exclusive shards, each limited to approximately 4000 concurrent players [110]. Despite the massive success of *WoW* and the huge revenue it is generating, WoW has at times struggled with scalability issues [132].

Genre	Game	Delay threshold
First Person Shooter (FPS)	*Counter-Strike*	150ms [48]
First Person Shooter (FPS)	*Unreal Tournament 2003*	150ms [14]
First Person Shooter (FPS)	*Quake 3*	180ms [10]
Sport	*Madden NFL Football*	500ms [103]
Real Time Strategy (RTS)	*Age of Empires*	500ms [16]
Real Time Strategy (RTS)	*Warcraft III*	500ms [122]
Massively Multiplayer Online Role Playing Game (MMORPG)	*Everquest 2*	1000+ ms [61]

Table 2.1: Delay requirements for smooth play in various games.

Shards rapidly reached their player limits, resulting in long queues of players waiting to join the shard. Several quests that result in a large number of players generating a large number of events have been known to crash the server. Our three proposed architectures, described in Chapters 3 to 5, each addresses the scalability of network games.

2.1.3 Responsiveness and Fairness

Delay in network games is measured as the Round Trip Time (RTT) from the client to the server and back (gamers call this *ping time*). A high network delay significantly degrades the player's enjoyment as it reduces the smoothness and responsiveness of the game [14, 105]. This frustrates players and impacts their performance [10, 103, 113, 135]. Further, beyond a genre specific threshold most games are unplayable. Table 2.1 lists the threshold at which the RTT delay becomes noticeable for various games.

Even if a player's delay is below the maximum threshold, if it is higher than his opponents' delays he is disadvantaged as his reaction time to game events will be longer [10]. As many multiplayer games are competitive this can be extremely frustrating. Delay fairness can be achieved by artificially inflating all players' delays to that of the slowest player [6, 59]. However, this solution is frustrating to the other players as the game's smoothness and responsiveness are reduced (consider the impact of a dial-up modem user connecting to a game involving players using broadband connections). Instead, commercial games use pipelining, interpolation, extrapolation, and event ordering algorithms to reduce the impact of network delay [135], increasing both responsiveness and fairness. However, these techniques cannot completely mitigate the disadvantage to a player of having higher delay than his opponents.

Pipelining updates increases responsiveness by allowing multiple updates to be in-transit simultaneously. For example, the *Source Engine* used in *Half Life* and *Counter-Strike* sends 20 updates per second to each client [135] (50ms between updates). When a client with 160*ms* delay receives update n, the server will already have transmitted updates $n+1$, $n+2$ and $n+3$. Pipelining does not improve the timeliness of updates, but does increase their frequency, improving responsiveness.

Avatar movement is typically smooth and continuous; however, in many games the server only sends updates at fixed intervals [14, 49, 122, 135]. Thus, avatar movement appears choppy and unrealistic. This problem is magnified when updates are late or lost. Interpolation and/or extrapolation smooth avatar movements, at the cost of increasing delay and state inconsistency respectively. Interpolation works by delaying the rendering of received updates until future updates have arrived. As the avatars' future positions are known, their movement can be rendered smoothly between these points; however, this adds additional delay between the server's and client's game states [135]. Extrapolation, also known as *dead-reckoning*, predicts an avatar's future location based on their direction and velocity. Avatars are rendered smoothly according to their predicted location until the next update is received. If the predicted location was incorrect the error is gradually corrected to provide smooth avatar movement. Extrapolation is very effective when avatar's movements are predictable; however, sudden unpredictable changes in direction cause significant inconsistency between the client's and server's game states [135].

If the RTT between a client and the server does not fluctuate wildly the server may compensate for this delay with an event ordering algorithm such as TimeWarp [42]. Note that the delay compensation technique used by the *Source Engine* [135] is a simplified version of TimeWarp. When an update is received the server rolls back the game state by half the RTT plus the interpolation delay, executes the update, and then re-executes all subsequent updates up to the present. TimeWarp is a boon for clients with high delay; however, as the server must store and re-execute previously received commands it increases the server's processing and memory requirements. Further, clients may perceive paradoxes, such as being shot by an opponent when hidden (shooting around corners), due to the delay in processing updates.

Our three proposed architectures, described in Chapters 3 to 5, each addresses responsiveness and fairness in network gaming. Note that for a game to be fair it must also address cheating, as discussed in Section 2.2.

2.1.4 Consistency

As communication through the Internet is not instantaneous inconsistencies occur between the game states of different hosts (clients, servers, and peers). Thus, each network game architecture requires an appropriate synchronisation mechanism to either prevent or gracefully recover from inconsistencies. As the server in C/S is trusted to order all game events and maintain the authoritative game state each client simply synchronises its state to that of the server, achieving consistency. However, achieving consistency in P2P is more difficult as there is no single clock for event ordering. While there are many possible solutions, two simple solutions are: (i) time is divided into rounds and the game cannot progress until all players have committed an update for that round [13], or (ii) a super-peer is elected that is responsible for ordering messages and storing the authoritative game state [62] (similar to the server in C/S). Note that solution (i) requires a mechanism for ordering events within a round (*e.g.*, by ascending player ID). While the focus of this thesis does not include consistency, our MRACS proposed in Chapter 4 addresses consistency between mirrors as it is highly relevant. Interested readers may refer to references [16, 65, 135] for a discussion of consistency issues and solutions.

2.1.5 Persistence and Reliability

MMOG are persistent online worlds, meaning the world continues to evolve when the player is offline. Player actions have an on-going impact on the world state, potentially impacting the game for months or years. Further, players expect the game to be reliable, and may wish to connect in at any time of day or night. Frequent downtime or crashes are frustrating for players, and may drive customers away [100]; therefore, running a reliable service is critical for maintaining a game's revenue stream.

Creating a persistent world with C/S is relatively simple as the server maintains the game state and should always be online. Further, if the server is well provisioned with high quality hardware, uninterruptible power supply, air conditioning, *etc.* it can achieve good reliability. However, as the server is a single point of failure, when it does fail it will take the entire system offline [36, 43, 52, 79]. As our proposed RACS architecture in Chapter 3 uses a single server, it also suffers from a single point of failure. However, by using mirrored referees, our MRACS architecture in Chapter 4 overcomes this problem.

As the reliability of individual peers in a P2P architecture is low, it is difficult to achieve persistence and reliability. However, if all peers are considered in aggregate the system may be far more reliable than C/S [36]. This would require building redundancy into the architecture's design such that the simultaneous failure of multiple peers would not disrupt the system [36]. Our proposed DRACS architecture in Chapter 5 does not directly address reliability or persistency. However, as DRACS distributes the authoritative game state across multiple referees we believe it can be extended to address these issues.

2.2 Game Cheating

2.2.1 Cheat Model

As opinions vary about what constitutes cheating in networked games there is no widely excepted definition. In this thesis we define cheating as *any user action that gives an advantage over his/her opponents that is considered unfair by the game developer*. We consider a cheater who can eavesdrop, replay, modify, delay, insert, and destroy messages sent and received by his host. He may also modify any program or software on his system including the game application, system functions, device drivers *etc*. Further, we assume a cheater has complete access to any information stored by the game application, including cryptographic keys and game state. We also assume groups of cheaters may collude to disrupt the system. We exclude general security issues such as authentication, denial of service, *etc.* as they are applicable to all Internet applications, and possible solutions are already documented in the literature.

2.2.2 Cheating Techniques

This section describes all known theoretical and practical methods players use to cheat. Where applicable, real-world examples are included.

Bugs: Bug cheats exploit design or implementation errors to gain an unfair advantage. Bugs do not require an in-depth knowledge/understanding of how/why they work, and do not require any additional programs or modifications to use. For example, the player ranking system in *Warcraft II* contained a designed error that gave rise to the win trading cheat; in which colluding players will repeatedly start matches against each other and then alternately surrender to give each of them the opponent victory points; thus, cheaters can climb to the top positions of the ranking ladder without playing any valid matches [150]. In Warcraft III win trading is prevented by randomising the participants when creating matches; however, this only works if the pool of players is large. An alternative solution is to include losses as well as victories when ranking players. An example of an implementation bug occurred in *Half-Life*, where a specific combination of actions allowed cheaters to reload weapons faster than honest players [111]; a significant advantage in FPS. This cheat was fixed by a software patch from the developer. For MMOG a database rollback to an earlier state may be required if the cheat seriously influenced the game world. Bugs are present in both C/S and P2P architectures and neither is resistant to this form of cheating. The accepted solution amongst the gaming industry and players for both C/S and P2P architectures is to release game patches to prevent the cheat; however, Delap *et al.* [47] argue that run-time verification may also be used to prevent bug cheats.

Real Money Transactions / Power Levelling: A Real Money Transaction (RMT) is when a player purchases a game item or virtual currency using a real-world currency [87]. Many Asian MMOG use RMT as their revenue model (free to play, but it costs money to purchase items); however, many western MMOG explicitly forbid RMT in their End User License Agreement (EULA), and the practise is considered cheating. Gold farming is a related phenomenon where low paid workers - usually in China - work full time playing MMOG to earn valuable items which are sold to players [87]. RMT occurs outside of the game, often on auction sites such as e-bay, or on dedicated message boards [131]. Most MMOG suffer from RMT; however, *WoW* is one of the most highly targeted games by gold farmers due to its massive player base. *WoW*'s publisher Blizzard regularly bans tens of thousands of players for gold farming in *WoW* [26]. While the method used by Blizzard to detect gold farmers is unknown, it is probable that they use statistical analysis of log files generated by the servers to detect RMT. As most P2P architectures distribute the game state and logic to peers, retrieving the required information to perform statistical analysis is far more difficult or impossible in P2P than in C/S.

A similar cheat is power levelling [87], where a cheater pays another person to play his character for her. This cheat is common in MMORPG where the objective is to gain experience and items. This form of cheating is an alternative business model for gold farmers; who offer this as a service.

Information Exposure: Also called secret revealing, the information exposure cheat results in the cheaters gaining access to information that they are not entitled to, such as their opponent's health, weapons, resources, troops, *etc.* [111]. This cheat is possible as developers often incorrectly assume that the client software can be trusted not to reveal secrets. Various methods used by cheaters to expose secret information include:

- Modifying the game client to directly expose secrets. For example, map hacks in RTS games such as *Warcraft III* [18].
- Using an external program to read data from the game client's memory. An example of this attack against *Age of Empires* was discussed by Pritchard [111].
- Modifying display drivers to render the world differently. For example, the *wall-hack* cheat in FPS games allows a cheater to see through walls by modifying the display drivers [124, 152].
- Sniffing game traffic as it passes across the network. One example of this cheat is *ShowEQ* [116], which captures *Everquest* traffic, interprets the packets, and displays the results to the cheater.

The most effective solution to prevent this cheat in C/S architectures is using On Demand Loading (ODL) [89]. Using this technique the server stores all secret information and only transmits it to the client when they are entitled to it. Therefore, the client does not have any secret information that may be exposed. P2P architectures can only use ODL if there is a trusted third party to store secret information; *e.g.*, trusted referees distributed to peers as proposed in Chapter 5.

Cheat Detection Systems (CDS) such as *PunkBuster* (PB) [50], *GameGuard* [104], *HackShield* [7], *Valve Anti-Cheat* [134], and *Warden* [46] are an alternative to ODL and prevent the first three forms of attack. The CDS operates similar to Anti-Virus software, scanning the player host's memory searching for cheating applications. The CDS matches check sums of running applications against a database of known cheats. However, as sniffing network traffic is entirely passive and does not take place on the cheater's computer it is impossible to detect packet sniffing using CDS or any other approach.

Bots/Aim Proxies/Reflex Enhancers: These cheats use computer generated input to control the player's avatar. Depending on the game genre, these cheats have one of two goals:

1. Use complex AI to automate repetitive tasks. One example of this is *WoW Glider* [95] .
2. Improve player's aim or reflexes. For example, automatically aiming at an opponent in an FPS [39].

This cheat is typically implemented by either:

- modifying the game client (*e.g.*, using *Ecstatic* for *Counter-Strike* [57]),
- running an external program to generate user input (*e.g.*, *WoW Glider* [95]), or
- routing player updates through a proxy server which modifies the player's commands (*e.g., Quake* aiming proxies [111]).

Both C/S and P2P architectures are vulnerable to this form of cheating. The first two techniques can be detected using CDS; however, there is no complete solution to prevent aim proxies. One possible solution to detect aim proxies is to use statistical analysis [153]; however, by introducing randomness into a bot's aim a cheater may go undetected [111].

Invalid Commands: Usually implemented by modifying the game client or data files, the invalid command cheat results in the cheater sending commands that are not possible with an unmodified game client. Examples include giving the cheater's avatar great strength or speed. Many games suffer from this form of cheating, including console games such as Gears of War [45]. The invalid command cheat is easy to prevent in C/S architectures as the server simulates and validates all commands, and can be trusted to produce the correct result. However, preventing invalid commands in pure P2P architectures is difficult as there is no trusted entity to verify player commands.

Mönch *et al.* [98] propose using tamper resistant techniques to prevent modifications to the game client; hence, preventing invalid commands. Their approach uses mobile guards; small segments of code downloaded from the game server that validate the game client using check sums and encrypting game data. Mobile guards are short lived; thus, there is insufficient time for an attacker to reverse engineer a mobile guard before it is expired. Although this does not prevent cheating, it can make it significantly more difficult. Furthermore, if successful this approach prevents some forms of information exposure and proxies/reflex enhancers. However, mobile guards require significant additional processing on the clients, and developing tamper proof software is a non-trivial task for the developer. We are not aware of any games using this technique; therefore, it is difficult to evaluate.

Suppressed Update: As the Internet is subject to packet loss most network games use extrapolation (dead-reckoning) to smooth player movements [13]. In the event of a lost/delayed update the server will extrapolate the player's movement from their current position, creating a smooth movement for all other players. Extrapolation usually allows clients to drop up to n consecutive packets (which are extrapolated) before they are disconnected. In the suppressed update cheat, a cheater purposely does not send up to $n-1$

consecutive updates, while still accepting opponent updates. Before the n^{th} update the cheater calculates the optimal move using the updates from their opponents and transmits it to the server. Thus, the cheater knows his opponent's actions before committing to his own, allowing him to choose the optimal action. Although we are not aware of any real world occurrences of this cheat, it is potentially possible for most FPS, and any game - either C/S or P2P - that uses extrapolation.

Architectures with a trusted entity, such as the server in C/S, prevent this cheat by making the server's extrapolated state authoritative. Players are forced to follow the extrapolated path in the event of lost/delayed updates. This gives a smooth and cheat free game for all other players; however, it will disadvantage players with slow or lossy Internet connections. As a slow or lossy Internet connection is already a major disadvantage [40] we believe this will not have a significant additional impact.

Cronin *et al.* [40] propose the Sliding Pipeline (SP) protocol to prevent this cheat in P2P architectures. In SP players constantly monitor the delay to their opponents and compare it with the timestamps of updates. Late updates indicate that a player is either suffering delay, or is cheating. The authors claim that this protocol will detect all cheaters, but acknowledge that players with poor connectivity may be falsely detected as cheaters (false positive).

Fixed Delay: This problem was discovered in *Madden NFL Football* by Nichols and Claypool [103], and was proposed as a method of cheating by GauthierDickey *et al.* [63, 64]. Fixed delay cheating involves introducing a fixed amount of delay to all outgoing packets. This results in the local player receiving updates quickly, while delaying information to opponents. For fast paced games this additional delay can have a dramatic impact on the outcome. This cheat is usually used in P2P games when one peer is elevated to act as the server; thus, it can add delay to all other peers. To prevent this cheat P2P games should use distributed event ordering and consistency protocols to avoid elevating one peer above the rest (See Section 2.3.3.2). Note, the fixed delay cheat only delays updates, in contrast to dropping them in the suppressed update cheat.

Inconsistency: Specific to P2P architectures, a cheater induces inconsistency between players by sending different game updates to different opponents. An honest player attacked by this cheat may have his game state corrupted, and hence be removed from the game by a cheater sending a different update to him than was sent to all other players. This cheat may also be used by a cheater or group of cheaters to gain an unfair advantage, and later merged with the other player's game state to make it undetectable [64].

To prevent this cheat updates sent between players must be verified by either a trusted authority, or a group of peers. In P2P protocols without a trusted third party the group must form a consensus about which updates are valid. The consensus is achieved by voting on the hashes of updates of all players; however, group selection is critical as several colluding cheaters could potentially bias the group vote [36].

Timestamp: This cheat is enabled as many games allow untrusted clients to timestamp their updates for event ordering. This allows cheaters to timestamp their updates in the past, after receiving updates from their opponents; hence, they can perform actions with additional information honest players do not have. C/S avoids this problem by using the arrival order of updates to the server for timestamping [64, 134]. Alternatively the proposal in [29] uses active RTT measurements between the server and peers to detect cheating in C/S architectures. See Section 2.3.3.2 for known solutions in P2P protocols.

Collusion: Collusion involves two or more cheaters working together (rather than in competition) to gain an unfair advantage. Colluding players often communicate via an external channel - over the phone, instant messaging, VoIP, *etc.* Collusion is extremely difficult or impossible to detect/prevent and has far reaching ramifications. There are many examples of collusion in networked computer games; however, one common example is of players participating in an all-against-all style match, where two cheaters will team up (collude) against the other players. This occurs in both C/S and P2P architectures, and is effectively undetectable. Yan [150] proposes several approaches to detect and prevent collusion including: using a web-cam to monitor opponents, artificial intelligence (AI), disabling chat features, rank tracking, log auditing, *etc.*; however, these methods are game specific, and cannot prevent sufficiently motivated players from colluding.

Spoofing: Spoofing is a traditional network security threat where a cheater sends a message masquerading as a different player [36]. For example, a cheater may send an update causing an honest player to drop all of their items. One potential obstacle for the cheater is to determine the victim's IP address, which is typically hidden in C/S games (but cannot be hidden in P2P games by definition). To prevent this cheat in both C/S and P2P, updates should be either digitally signed or encrypted. With either technique the receiver can validate the sender's identity. We are not aware of any real-world games where this has occurred.

Replay: If a cheater receives digitally signed/encrypted copies of an opponent's updates he may be able to disadvantage an opponent by resending them (replay) at a later time [36]. As the updates are correctly signed or encrypted they will be assumed valid by the receiver. To prevent this in C/S and P2P, updates should include a nonce (unique number), such as a round number or sequence number. When an update is received the receiver checks to ensure the nonce is fresh (has not been used before). While many games are vulnerable to replay attacks, we are not aware of any examples where this cheat has been used.

Blind Opponent: A cheater may purposely drop updates to opponents, blinding them about the cheater's actions, while still accepting updates from opponents [145]. This cheat is only possible in some P2P protocols [108]. A tit-for-tat scheme where players stop sending updates to cheaters - effectively blinding the cheater as well - is an insufficient solution for this cheat as there are instances where dropping updates would still give the cheater an advantage, such as if they need to make a retreat. We are not aware of any real world instances where this cheat has been used. P2P solutions are discussed in Section 2.3.3.2.

Undo: Some P2P protocols [13, 36, 64] use a commit/reveal scheme to prevent the suppressed update, fixed delay, timestamp, and blind opponent cheats; however, if the reveal step is not enforced (as in [36, 64]) it is possible for a cheater to reveal their opponent's move and assess it, before deciding if they will reveal their move. If a cheater does not reveal his move he effectively undo the move. P2P protocols that require all updates to be revealed (*e.g.*, Lockstep and AS) or do not use the commit/reveal process are immune. This cheat was first discussed in [145].

This thesis focuses on the information exposure, invalid command, suppressed update, timestamp, fixed delay, inconsistency, spoofing, replay attack, undo, and blind opponent cheats. Thus, our cheat prevention/detection is equal to that in C/S. Other cheats are treated as in C/S, *i.e.,* bug cheats are prevented by patches released by the developer, RMT and power levelling are detected using statistical analysis of log files, bots/reflex enhancers are detected using a CDS, and collusion and proxies/reflex enhancers cannot be detected or prevented.

2.2.3 Cheat Classifications

The first published review of cheating and cheat detection/prevention was done by Matt Pritchard [111], one of the developers of *Age of Empires*. It includes his experiences both as a developer and player of computer games. This industry focused article discusses specific real-world cheats, and covers practical methods to discourage them. Pritchard acknowledges that many of the solutions presented do not prevent cheating, but make it far more difficult for players to cheat. He argues that if the difficulty of cheating is greater than the difficulty involved in playing the game players will not cheat.

Following this industry focused article, Yan [150] provides a theoretical review of cheats, particularly focused on online Bridge. Of particular interest is his discussion of preventing collusion between players. While there are several possible counter-measures, for all practical purposes it is impossible to completely eradicate collusion in online games.

Kabus *et al.* [77] divide malicious players into cheaters and griefers. A cheater actively attempts to break the game rules to gain an unfair advantage; whereas, a griefer attempts to hurt other players' game experience without gaining any advantage. Griefers can be extremely difficult to combat as they are not motivated by self interest; for example, a griefer may attempt to crash the server, even though this means they too will be unable to play the game. To annoy other players griefers may break game rules (*e.g.,* corrupting the server's authoritative game state), or simply abuse game features (*e.g.,*., using the in-game chat features to verbally abuse other players). The main focus of this thesis is to detect / prevent cheating. However, our solutions in Chapters 3 to 5 also prevent griefers damaging the authoritative game state. As griefing using game features does not break any game rules it is beyond the scope of this thesis.

Yan and Randell [151, 152] provide an extensive list of cheating techniques, and formed a taxonomy with regard to the underlying vulnerability (*what is exploited?*), consequence (*what type of failure can be achieved?*) and the cheating principle (*who is cheating?*). The authors find that traditional methods of security in software - confidentiality, integrity, availability, and authenticity - are necessary but insufficient to defend against cheating. Although large and detailed in its taxonomy, their characterisation of cheating lacks structure, and it is argued that new forms of cheating cannot be easily integrated [101]. Note, we do not include several categories of cheating proposed by Yan and Randell (*i.e.*, cheating by denying service to peer players, cheating by compromising passwords, cheating by exploiting lack of authentication, cheating by compromising the game server, cheating related to internal misuse, and cheating by social engineering) in our classification in Section 3.1, as they are relevant to all secure network applications and not directly related to game mechanics; hence, we believe they are general security issues, not cheating issues.

Neumann *et al.* [101] distinguish three categories of cheating based on the threatened game property: confidentiality, integrity, and availability. Confidentiality requires that a cheater cannot access state information he is not entitled to (*i.e.*, secret information), else he will have an unfair advantage when selecting what actions to take. Integrity ensures that a cheater cannot modify the game state unfairly; hence, the integrity of the game state is maintained. Availability requires that the entire game is available to all players at all times, and that cheaters cannot prevent the game from progressing fairly. Unfortunately, this paper only provides a brief coverage of cheating; hence, the authors only briefly discuss possible cheats and their methods of attack.

GauthierDickey *et al.* [63] proposed a cheat classification scheme comprising four levels: game, application, protocol, and network. Game cheats do not require any external programs or modification and occur entirely within the game; application cheats require using or modifying applications; protocol cheats interfere with the game's communication protocol; and network cheats involve network security issues. The authors consider nine forms of cheating, *i.e.,* denial of service in the network level; fixed delay, timestamp, suppressed update, inconsistency and collusion in the protocol level; secret revealing (also called information exposure) and bots/reflex enhancers in the application level; and breaking game rules (also called bugs) in game level. Corman *et al.* [36] extend this classification with two additional protocol level cheats: spoofing and replay attacks. This classification forms a strong foundation for organising cheats; however, it is slightly too narrow to accommodate all forms of cheating. In particular, this classification can only accommodate information exposure and proxies/reflex enhancers at the application level, ignoring other attack vectors. Further, their classification does not include the RMT/power levelling, invalid command, undo, and blind opponent cheats. In Section 3.1 we extend their classification [64] to address these shortcomings.

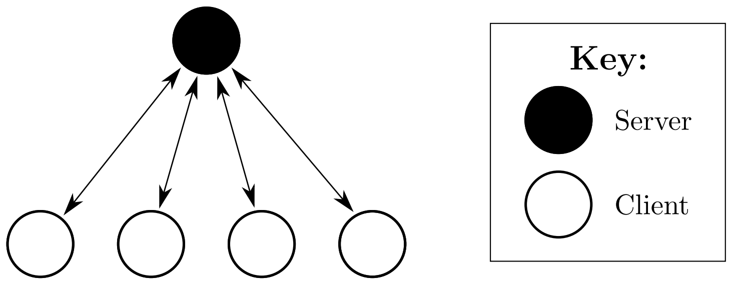


Figure 2.1: Client/Server architecture.

2.3 Network Game Architectures

2.3.1 Client/Server (C/S) Architecture

The C/S architecture shown in Figure 2.1 is comprised of many clients communicating through a single server. The server is the game authority whose tasks include: (i) receiving player updates, (ii) simulating game play, (iii) validating and resolving conflicts in the simulation, (iv) disseminating updates to clients, and (v) storing the authoritative game state. Additional tasks such as storing offline players' avatar state, downloading joining players' avatar state, authenticating joining players, and billing, are typically handled by auxiliary servers. As these tasks do not influence the game's scalability, responsiveness, fairness, *etc*., they are not addressed in this thesis.

By using a single trusted server addressing cheating, consistency, conflict resolution, and persistency issues is simplified. The server is trusted to validate player actions, simulate game play fairly, and keep secret information confidential [70, 77]; thus, preventing cheating. With one authoritative game state, clients need only synchronise with the server to maintain consistency [135]. As the server is always online, C/S is inherently persistent [70]. Finally, as the clients and server do not require complex synchronisation, validation, and anti-cheat protocols, it is the easiest architecture to implement [70, 149].

However, with only one trusted server, C/S has poor scalability, responsiveness, delay fairness, and reliability. As the server must receive all client updates, simulate all updates, and transmit new game state to all clients in real time, it is a bandwidth and processing bottleneck, limiting scalability [13, 43, 64, 67, 108]. For a C/S MMOG to support thousands of clients the server must be comprised of a cluster of hosts acting as one logical unit. As mentioned in Section 2.1.2, to support hundreds of thousands or millions of players requires sharding the game's universe into many independent virtual worlds. Each shard run by a separate server comprised of many hosts. With the exception of EVE Online [24], all commercial MMOG require sharding to support their player populations.

As all updates are routed through the server, delay is not optimal [64]; and if interacting players have high client-to-server delay, but low delay between each other, responsiveness is far from optimal. Further, players with low client-to-server delay have better responsiveness than those with high client-to-server delay [25]. This gives them an unfair advantage.

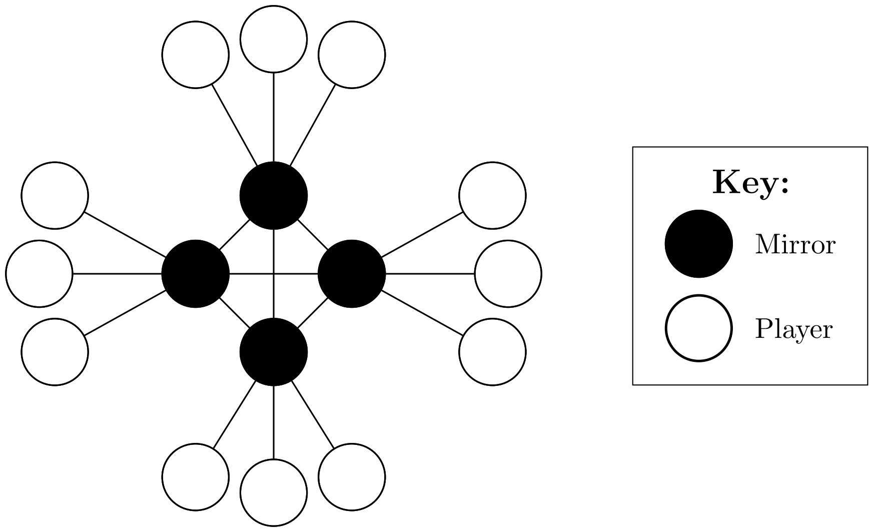


Figure 2.2: Mirrored Server architecture.

Finally, while a well provisioned server has far better reliability than its clients, the server is a single point of failure, potentially halting the entire game [43].

Despite its limitations C/S is the dominant network game architecture, and to the best of our knowledge it is used by all commercial MMOG, including the genre dominating *World of Warcraft* (*WoW*) [19].

2.3.2 Mirrored Server (MS) Architecture

2.3.2.1 Architecture

The MS architecture [41–43] shown in Figure 2.2 comprises multiple trusted servers (mirrors) deployed at geographically different locations connected via a private well-provisioned (low delay, high bandwidth, multicast enabled, lossless) network. Each mirror has its own Internet connection, and clients connect to their closest mirror. Each client sends every update to its connected mirror (ingress mirror - I-mirror) which, in turn, broadcasts it to all other mirrors (egress mirrors - E-mirrors). Then, all mirrors simulate the game world based on all client updates and, therefore, are able to resolve state inconsistencies. Finally, every mirror periodically sends updates to its clients.

Moving the server processing into the network - closer to the clients - reduces delay, distributes the bandwidth cost throughout the network, and removes the single point of failure. However, player updates in MS, like in C/S, are routed through mirrors; increasing delay, reducing fairness, and consuming the mirrors' bandwidth and processing power. There are several proposals to improve MS [69, 106]; however, they focus on fair and interactive event delivery.

The federated peer-to-peer architecture [118] is similar to MS as it uses multiple publisher provisioned hosts (called reflectors) to distribute updates among clients. Clients connected to different reflectors communicate via messages forwarded between reflectors, similar to MS. To minimise the reflectors' processing requirements they do not simulate the virtual world, unlike the mirrors in MS. Without a trusted authority to validate player updates, this architecture is vulnerable to cheating.

2.3.2.2 Synchronisation

Updates exchanged between mirrors may be received at different times due to the transmission delay in the private network, resulting in inconsistencies among mirrors. One solution is Bucket Synchronisation (BS), in which time is divided into buckets, and all updates in each bucket occur simultaneously. The execution time for each bucket is delayed by Δ so that all updates for each bucket are received, synchronising all mirrors. If a late update arrives after its bucket has been executed, the update is added to the next bucket. However, if a bucket contains multiple updates from a player, due to a delayed update from an earlier bucket, only the newest update is executed. If a bucket does not contain an update for a player, the player's state is extrapolated. The advantages of BS are: (i) it is simple to implement, (ii) it has low memory overhead, and (iii) it has low processing overhead as each update is only executed once by each mirror. However, as MS requires all updates to be sent via the mirrors the additional Δ delay reduces responsiveness, making MS with BS mirror synchronisation unsuitable for fast paced games

To support fast paced games Cronin *et al.* [43] proposed Trailing State Synchronisation (TSS). Each mirror maintains l states of the game world: S_0, S_1, S_2, S_3, ..., S_{l-1}, where S_0 is the leading state and is immediately sent to the players for fast responsiveness, while the $l-1$ trailing states are used to resolve inconsistencies. Each TSS state has increasing delay behind the wall clock time. When an E-mirror receives an update from another mirror the update is simulated in all states newer than the update time. If the simulation results in two inconsistent states, then a rollback occurs, else the simulation is allowed to continue. To rollback, the trailing state is copied to the leading state, dynamic memory structures are repaired, and all elapsed updates are re-executed. TSS is only effective when simulating updates is inexpensive as each update is executed at least l times by each mirror for TSS with l states, and more when rollbacks occur. Furthermore, performing a rollback is expensive, due to memory management costs and the need to re-execute many updates. Thus, the processing bottleneck in MS is worse than that in C/S. In addition to the extra processing overhead, TSS requires l times the memory of BS to store each copy of the game world, and players may perceive strange temporal anomalies when rollbacks occur. Note that TSS with a delay Δ in the leading state and no trailing states is in essence BS.

2.3.2.3 Security

As MS utilises trusted mirrors, it is possible to achieve the same level of security as in C/S; however, the protocol in Cronin *et al.* [43] is vulnerable to time-stamp cheating because updates are time-stamped (for event ordering) by the untrusted clients. Consider a mirror

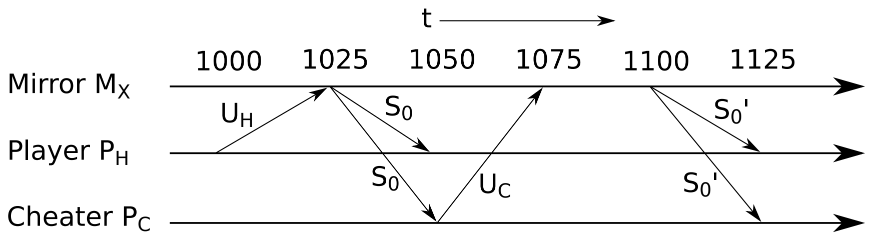


Figure 2.3: Timestamp cheating in the MS architecture.

M_X with two connected players P_H and P_C (a cheater) with $25ms$ delay, and two states $S_0 = 0ms$ and $S_1 = 100ms$. As shown in Figure 2.3, P_H sends a shoot update U_H at $t = 1000ms$ to M_X. At $t = 1025ms$ M_X receives and simulates U_H in S_0, calculates a hit against P_C, and sends the new state S_0 to P_H and P_C. Player P_C finds he has been shot and cheats by sending a dodge update U_C, with a false timestamp of $975ms$ at $t = 1050ms$. Receiving U_C, M_X executes it in S_0 and S_1 at $t = 1075ms$. At $t = 1100ms$, M_X executes U_H in S_1 and detects an inconsistency with S_0 (miss vs. hit). Thus, M_X performs a rollback and notifies P_H and P_C of the result S_0' (a miss). Note, this cheat is prevented in MRACS by requiring each I-mirror to timestamp updates (Discussed in Section 4.1.4).

2.3.2.4 Client-to-Mirror Assignment (CMA)

In MS [41–43] each joining client is assigned to its closest mirror to minimise game delay. This greedy approach is possible if each mirror has sufficient capacity for all connecting clients. However, in reality, mirrors have finite resources that support only a limited number of players. As players are not uniformly distributed, some mirrors may be overloaded; thus, for a mirrored system with fixed resources (*e.g.*, MS), a Client-to-Mirror Assignment (CMA) algorithm is needed to maximise game responsiveness. We formally define our CMA problem in Section 4.2.

For C/S games, server selection is performed by the client software, which probes a list of servers and ranks them according to delay and configuration [11, 12] (*e.g.*, number of players, map, special rules, *etc.*). The player then selects a favourable server from this list. This approach is insufficient for CMA as it does not allow reassigning clients due to the mirrors' workload changes. Claypool [34] investigated group server selection, in which a group of players select a server such that the client-to-server delay for all players is below a threshold. It is obvious that as the number of players increases, the number of potential servers decreases. The results [34] showed that for FPS it is difficult or impossible to select a server that will satisfy a group of seven or more players. Using mirrored servers is a possible solution as each player need only connect to his closest mirror.

The Clients-to-Servers Assignment (CSA) problem for web servers and DNS systems [53] is similar to CMA. However, CSA does not consider reassigning clients when server workload changes as it assumes short-lived sessions. Further, solutions using round robin DNS [53] cannot reassign clients as IP addresses are cached. In contrast, a network game session may last for several hours; thus, CSA solutions are not applicable for CMA.

Lee *et al.* [88] and Ta *et al.* [128] proposed two optimisation problems involving assigning clients to mirrors/servers, similar to our CMA. Lee *et al.* [88] assume a large number of servers, each simulating only a portion of the virtual world. Further, the virtual world is divided into many zones, and each zone's current state is mirrored across one or more servers. Their goal is to maximise the number of clients connected within a delay bound and to minimise mirror processing by reducing the number of mirrors simulating each zone. The authors [88] showed that this problem is NP-hard, and proposed a heuristic greedy algorithm to produce near optimal results.

Ta *et al.* [128] divided the virtual world into zones containing interacting clients and considered a set of geographically distributed and well-connected servers. A target server is assigned to each zone for simulating all events in the zone; *i.e.,* each zone is simulated by only one server and is not mirrored. Each server may simulate multiple zones but has limited CPU capacity. The contact server for a client is the server to which it is connected. If a client's contact and target servers are different, the contact server is responsible for forwarding updates between the client and the target server. Their goal was to maximise the number of clients with client-to-server delay below a threshold subject to CPU capacity constraints.

The problems addressed by Lee *et al.* [88] and Ta *et al.* [128] are related but fundamentally different to our CMA problem in Section 4.2 as the goals and assumptions are different. Further, their solutions do not consider, in real time, new client joins, existing client leaves, and clients moving between zones. Thus, the average delay of the remaining players may increase when some players leave. Note that for optimal results the mirror placement problem must also be addressed [117]. However, as mirror placement is heavily influenced by business considerations, this problem is beyond the scope of this thesis [100].

Our CMA solution in Section 4.2 is a modification of the Terminal Assignment (TA) problem in *teleprocessing network optimisation*, which is equivalent to the *transportation problem* in operations research [22, 129]. The following description is a modification of that in Kershenbaum [81]. The TA problem is modelled as a bipartite graph matching problem between a set of terminals, T, and concentrators, C. The cost of connecting a terminal $T_i \in T$ to a concentrator $C_j \in C$ is c_{ij}. Each concentrator C_j has capacity W_j, and each terminal T_i requires w_i units of capacity at a concentrator. The objective is to minimise the cost, min z, of connecting all terminals to concentrators, subject to the concentrator's capacity constraints. Formally:

$$\min z = \sum_{T_i \in T, C_j \in C} c_{ij} x_{ij} \tag{2.1}$$

subject to:

$$\sum_{C_j \in C} x_{ij} = 1 \qquad \forall T_i \in T \tag{2.2}$$

$$\sum_{T_i \in T} w_i x_{ij} \leq W_j \qquad \forall C_j \in C \tag{2.3}$$

$$x_{ij} \in \{0,1\} \tag{2.4}$$

where x_{ij} is 1 if T_i is assigned to C_j (denoted as $T_i \to C_j$). Equation (2.2) guarantees that all terminals are assigned to a concentrator, and Equation (2.3) ensures that no concentrator is overloaded. Note, there are several variations of the TA problem such as allowing one terminal's resource requirements to be shared by multiple concentrators, or requiring each terminal to be connected to multiple concentrators for redundancy. The problem as defined above is NP-complete [81, 82]; however, the special case where all terminals have equal requirements ($w_1 = w_2 = ... = w_n$) is solvable in polynomial time using the Sequential Assignment (SA) [129] or Alternating Chain [81] algorithms. We describe the SA algorithm as it is used in Chapter 4.

The SA algorithm iteratively assigns one terminal to a concentrator. Thus, at the start of iteration k, terminals T_1, T_2, ..., T_{k-1} have been optimally assigned. Note, SA requires that at each iteration the assignment is optimal; else, it may loop infinitely. There are two possible cases when assigning T_k:

(i) the concentrator closest to T_k, concentrator C_l ($c_{k,l} \leq c_{k,j}, \forall C_j \in C$), has spare capacity; or

(ii) concentrator C_l does not have spare capacity.

For case (i), T_k is assigned to C_l as this is the optimal assignment, and SA continues to the next iteration. For case (ii), a sequence of terminal reassignments (called a chain) may be required to optimally assign T_k, or T_k may be assigned to a concentrator with spare capacity. For each concentrator C_f at full capacity, the SA algorithm calculates the lowest cost chain of assigning T_k to C_f using the labelling procedure below, and selects the lowest cost chain. If there exists a concentrator C_j with spare capacity and c_{ij} equal to the lowest cost chain, T_k is assigned to C_j; else, the lowest cost chain of reassignments is performed.

Before initiating the labelling procedure the *transfer distance, d_{jl},* between all pairs of concentrators C_j and C_l ($j \neq l$) must be calculated:

$$d_{jl} = \begin{cases} \infty & j \text{ is empty} \\ \min(c_{il} - c_{ij}), \quad \forall T_i \to C_j & \text{otherwise.} \end{cases}$$

Note that d_{jl} can be negative. The labelling procedure works as follows. The label for each concentrator C_j is comprised of two parts: $L_j(cost)$, the cost of the chain at C_j; and $L_j(next)$, the next concentrator in the chain. Initially $L_j(cost) = \infty$ and $L_j(next) =$'-' if C_j is full; else, $L_j(cost) = 0$ and $L_j(next) =$'-'. For every full concentrator C_f, if there exists a lower cost chain $L_j(cost) + d_{fj}$, update the label to be $L_f(cost) = L_j(cost) + d_{fj}$ and $L_f(next) = C_j$. This is repeated until no labels are changed. All chains are completed by adding the cost of assigning T_i (*i.e.*, $\forall C_j \in C, L_j(cost) = L_j(cost) + c_{ij}$).

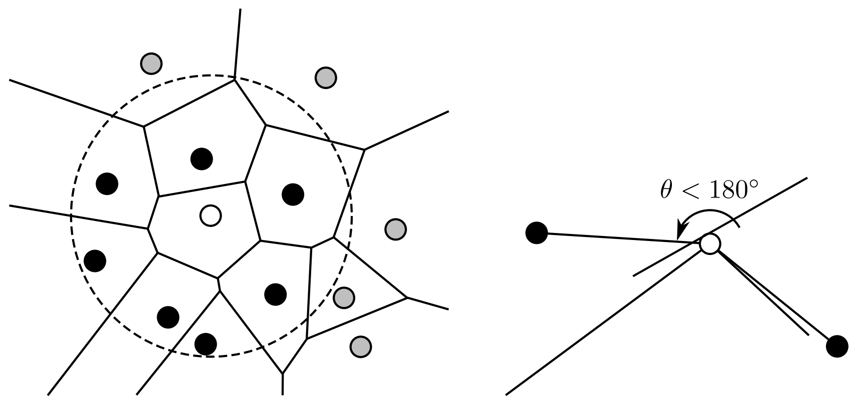
2.3.3 Peer-to-Peer (P2P) Architectures

In P2P applications all hosts make requests of each other, and provide services to each other; thus, all hosts are considered equal (*i.e.,* peers). P2P systems are extremely scalable as there is no central bottleneck, and the system is resource growing; *i.e.,* as peers join and make requests the number of peers able to service requests increases. Further, peers can request services of those with low peer-to-peer delay, maximising responsiveness. However, as peers have a high probability of failure (compared to the server in C/S), join and leave without warning, and may act maliciously, ensuring the correct data is received requires protocols far more complex than those in C/S. The most common use of P2P is in file sharing applications, of which BitTorrent is the most common [35, 112]. BitTorrent works by dividing large files into small pieces which peers exchange between each other. Peers both request pieces from other peers, and service requests made by other peers.

The IP protocol used in the Internet provides only the most basic routing and reliability services for network applications. To simplify the development of complex P2P applications such as games, network overlays involve peers connecting in either a structured or unstructured manner to provide high level services for applications such as searching/data location, multicasting, reliable data storage, trust/authentication, *etc.* [90].

Distributed Hash Tables (DHT) are network overlays that provide $\{key, value\}$ lookups within a bounded number of hops (often $O(log(n))$, where n is the number of peers in the DHT). Many DHT also provide auxiliary functionality such as reliability guarantees and application layer multicasting. The services are provided using a single pre-defined key space. Each peer selects or is assigned an ID (NodeID) from within the key space which determines its location in the DHT, and its neighbouring peers (peers it connects with to maintain the DHT). The *values* (the data to be stored) are deterministically assigned a *key* from within the key space. The *value* is stored by the peer whose ID is closest to the *key*. Each DHT provides a routing mechanism such that the *key* can be used to locate the storing peer and the *value* retrieved. Many different DHT have been proposed such as *Chord* [125], *Pastry* [119], *Can* [115], *Tapestry* [156], *etc.* See reference [90] for a review of P2P network overlays.

In addition to the high level features offered by network overlays, delay sensitive P2P applications require the ability to locate peers with low delay in a timely manner. As P2P systems often include millions of peers (*e.g.*, [51, 126]) it is infeasible in terms of time, bandwidth, and storage to measure the n^2 delays between all pairs of peers. Delay estimation schemes such as IDMaps [60], GNP [102], and Vivaldi [44] have been proposed to estimate the n^2 delays using only $O(n)$ measurements and storage. While these schemes are not as accurate as direct measurement, they do allow peers with low delay to be located efficiently.



(a) Each peer in VON generates a Voronoi diagram and connects to all of its enclosing neighbours (solid black circles) [71].

(b) Each peer in Solipsis must maintain connectivity with several neighbours such that the angle θ between all pairs of adjacent avatars is below π radians [80].

Figure 2.4: Maintaining global connectivity using avatar location.

2.3.3.1 Architectures Without Cheat Detection/Prevention

There have been a great number of proposals for P2P network game architectures using a wide range of techniques. Unfortunately, the vast majority do not directly address cheating, tacking on anti-cheat measures or ignoring cheating entirely. As cheat prevention is critical for MMOG, these proposals are not suitable for real-world deployments.

Knutsson *et al.* [84] proposed *SimMud*, a P2P network game architecture for MMOG. The virtual world is divided into zones, and the state of each zone is controlled by a coordinator peer. The coordinator resolves inconsistencies among players, and acts as the root of the multicast tree to disseminate game state. *SimMud* makes heavy use of the *Pastry* DHT to provide coordinator look-up and application layer multicasting. The use of a DHT greatly simplifies *SimMud*'s design and implementation; however, it also adds significant delay, making *SimMud* inappropriate for many genres of games. *Zoned Federation* [74], *MOPAR* [154], and *Colysius* [120] use a DHT for global connectivity similar to *SimMud*; however, the DHT is only used to locate peers. All other communication is unicast between nodes, significantly reducing delay. Note that these schemes [74, 84, 120, 154] do not address cheating, and the use of a DHT to provide global connectivity provides a new attack vector for malicious users.

An alternative approach to maintain global connectivity is to establish P2P connections corresponding to the location of players' avatars in the virtual world. The *VON* [71] architecture uses Voronoi diagrams generated from avatar locations, and requires every peer to connect to its enclosing neighbours as shown in Figure 2.4a. *Solipsis* uses a similar approach, but requires the angle θ between avatars to be below π radians, as shown in Figure 2.4b. In both VON and Solipsis neighbour discovery is achieved by communication with connected peers.

For a peer with low bandwidth, such as a player using a dial-up modem, it may be unable to communicate its updates to other peers in a timely manner. This is especially true if the peer is acting as a region controller, and must forward updates from all peers. A proposed solution is to use End-System-Multicast (ESM) to overcome this problem, by having high-bandwidth peers forward updates for low-bandwidth peers. The DHT in *SimMud* provides ESM capability, but is not optimised to minimise delay; therefore, it is not suitable for games with tight delay bounds. Further, as *SimMud* does not restrict the task of multicasting updates to high bandwidth peers, a low-bandwidth peer may be overburdened. The ESM scheme proposed in Donnybrook addresses these issues, as only peers with good connectivity (high bandwidth, low delay) may act as multicasters [17]. Further, the multicast tree is restricted to a height of 2 to reduce the delay introduced and minimise the cost of tree maintenance.

The N-tree protocol [62] provides reliable event ordering while using ESM. The virtual world is divided into regions, and each region is a node in the multicast tree. Each region may be further subdivided into a sub-tree, with the existing node acting as the parent. The root of the tree encompasses the entire virtual world. Peers subscribe to leaf nodes according to their AoI. Within each leaf node peers use a total event ordering protocol requiring $\Omega(m^2)$ messages, where m is the number of players in the leaf, to maintain consistency. When an event occurs within a leaf that exceeds the leaf's region, the event is propagated to the node's parent recursively, until a node which entirely encompasses the event is found. The encompassing node uses the event's timestamp to maintain event ordering, and uses the tree to multicast the event to all descendants. The N-tree protocol is successful in correctly ordering events and minimising the number of superfluous messages that are exchanged. However, the additional delay introduced by the protocol may make it unsuitable for many genres of games.

The Peer-to-Peer with Central Arbitrator (PP-CA) architecture is a hybrid between C/S and P2P. Peers exchange updates directly to minimise delay and reduce the server's bandwidth (the central arbitrator). Peers send a copy of each update to the server; which is used to maintain the authoritative game state. Inconsistencies between peers are resolved using the authoritative state. While PP-CA does increase game responsiveness and reduce the server's bandwidth, allowing peers to exchange updates enables the inconsistency, fixed delay, and blind opponent cheats, which are not addressed by the proposal. As discussed in Section 3.4.1, our RACS protocol prevents or detects these cheats.

2.3.3.2 Architectures With Cheat Detection/Prevention

There have been many proposals to address cheating in P2P architectures; however, most address only a subset of cheats, making them inadequate for real-world use. In particular, many proposals do not address the information exposure or invalid command cheats, which are regularly used in the real world.

Kabus *et al.* [77] discuss three different techniques that may be used in P2P architectures to prevent/detect cheating: mutual checking, log auditing, and trusted computing. The principle of mutual checking is that *"you may not trust a single client, but you trust the consensus of multiple unaffiliated clients*"; therefore, multiple randomly selected clients can be trusted to validate player actions before the game state is modified, preventing cheating. The second approach, log auditing, does not prevent cheating, but allows the game to detect cheating when it has occurred - albeit much later in some games. When cheating is detected the game performs a rollback to undo the effects of the cheat. Log auditing is not appropriate for all forms of games, particularly MMOG that do not have an end state (game completion). The final solution, trusted computing, involves using special hardware that prevents cheaters from modifying the game or running cheating programs. This solution is currently inappropriate for PC users; however, it is being actively used in console games. Unfortunately, several trusted computing solutions have been shown to contain weaknesses allowing players to run untrusted programs, enabling cheating [77].

The Lockstep protocol prevents the fixed delay and timestamp cheats [13]. Game time is divided into rounds, and each round consists of a commit phase and a reveal phase. In the commit phase every player calculates their update and sends a cryptographically secure one-way hash of the update to all other players. Once all players have exchanged hashes the commit phase ends and the reveal phase begins. Each player transmits his updates to all other players, and every received update is validated against the corresponding hash. If the hash does not match, the update is discarded. As players do not receive updates from their opponents until after they have committed to their update (sending the hash) the fixed delay and timestamp cheats are prevented. However, assuming d is the delay between the two slowest players, the minimum round length is $3d$ as each player must: (i) commit his hash to all peers; (ii) wait for acknowledgement from all peers; and (iii) receive all peer updates. Therefore, Lockstep has poor responsiveness, making it inappropriate for fast paced games.

Asynchronous Synchronisation (AS) was proposed by Baughman *et al.* [13] to increase the responsiveness of Lockstep. In AS players progress rounds at their own rate, and only enter Lockstep communication with players they are interacting with (players with overlapping AoI). Therefore, the game progresses as fast as the slowest player among a group of interacting players. Further, the round length is bounded, and any opponent that cannot send messages within the round is removed from the game. While AS is considerably faster than Lockstep, it is still too slow for many genres of games as the minimum round length is still $3d$.

The New Event Ordering (NEO) protocol [64] is an evolution of AS, designed to increase responsiveness and prevent cheating. Similar to Lockstep, NEO divides time into rounds of length d. Each player sends an encrypted update to all other players in each round (players commit to an update), and sends the decryption key in the following round. Thus, the playout latency of NEO is $2d$. An update in NEO is only considered valid if the majority of peers receive the update on time (within the same round). When peers reveal their

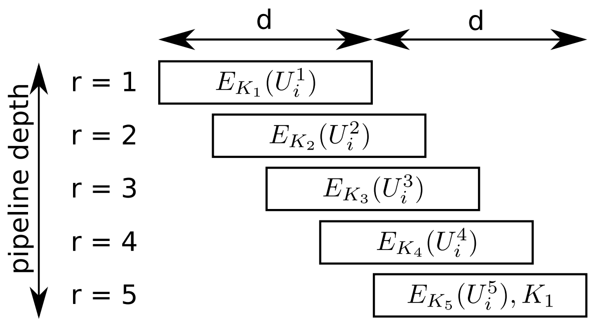


Figure 2.5: Rounds pipelined in NEO.

encryption keys in the following round they also exchange votes to establish which updates should be considered valid. Updates that are not received on-time by the majority of peers are discarded. Using a majority vote rather than a consensus allows the game to continue in the event of packet loss, unlike Lockstep and AS. However, if a player cannot communicate with the majority of his peers within d, he is removed from the game. As NEO is tolerant of packet loss it uses unreliable communication; this increases responsiveness as the decryption keys are transmitted without waiting for the corresponding committed updates to be acknowledged.

To increase responsiveness the players in NEO vote to increase/decrease d. The maximum value for d is specified by the developer, and is the maximum round length such that the game remains playable. Periodically players vote to either increase, stay the same, or decrease the round length. If the majority vote is to increase, d is doubled (up to the maximum value); if the majority vote is to decrease, d is reduced by $20ms$. This *Multiplicative Increase/Additive Decrease* (MIAD) scheme is similar to that used in TCP/IP congestion avoidance. The advantage of this scheme is that it is fully distributed; however, it requires more bandwidth, is slower, and less accurate than our centralised approach discussed in Section 3.3. Without a trusted authority to tally the votes, all votes must be sent to all other players requiring $O(n^2)$ messages, where n is the number of players in the group. Further, as it requires several rounds of adjustment before the round length converges, this approach is slow. Finally, as the round length adjustment is very coarse (either double or reduce by $20ms$), the result will only approximate the optimal value.

To further improve the responsiveness, NEO allows rounds to be pipelined as shown in Figure 2.5. Note, message $E_{K_r}(U_i^r)$ is the update U_i^r by player P_i for round r, encrypted using key K_r. As in the basic NEO protocol, the key for a message is only sent after the round is complete, preventing cheating. Pipelining simply allows the process to occur in parallel.

The authors argue that NEO is secure against the fixed delay, timestamp, suppressed update, and inconsistency cheats; however, Corman *et al.* [36] showed that the cryptographic techniques were used incorrectly, allowing three different cheats:

(i) An attacker can replay updates for another player,

(ii) An attacker can construct messages with any previously seen votes attached. Since the votes are signed the messages will appear to come from another player,

(iii) An attacker can send different updates to different opponents.

The authors propose the Secure Event Agreement (SEA) protocol [36] to address these problems. This protocol is identical to NEO, but applies cryptographic techniques differently to prevent cheating. The techniques also provide security against replay and spoofing attacks, and have lower processing overhead than those used in NEO.

As previously mentioned, the sender of each message includes votes for the updates it received in the previous round. To prevent the inconsistency cheat the vote is the hash of the received update. The majority of peers must have the same hash for an update to be accepted as valid. By including the hashes of all received updates each message grows in size by $O(n)$, and must be sent to all $n-1$ peers. Thus, NEO and SEA require $O(n^2)$ bandwidth per player, severely limiting the maximum size of n, and NEO/SEA's bandwidth scalability.

To reduce the required bandwidth peers are divided into groups of interacting players, and use NEO/SEA independently for each group. However, as the majority vote determines the validity of updates, the majority of players in a group must be honest; else, a group of colluding players could cheat [36, 37, 64].

To prevent collusion a secure verification group can be elected to control the game state for each group of interacting players [37]. Players send their updates to all members of the verification group, which use SEA to vote on the valid game state. The Secure Group Agreement (SGA) protocol [37] is a distributed approach to securely select the verification group. From the set of players in a game, SGA randomly selects the verification group, such that the probability of the majority of peers colluding is below a predefined limit. Combining SGA with SEA prevents collusion; however, as SGA ignores the network location of peers the verification group will be distributed randomly throughout the Internet, removing the delay benefit of P2P over C/S. From the paper [37] it is unclear whether players will receive the game state from the verification group, or if peers will also exchange updates.

Kabus and Buchmann [76] propose a scheme similar to using a verification group, but its design minimises delay at the expense of bandwidth. We refer to this scheme as P2P-RC. The virtual world is divided into zones and player actions are confined to one zone. Each zone has several Region Controllers (RC), which act as the verification group. Each update generated by a peer is sent to all RC, which simulate the update and return the resulting game state (a vote). The peer tallies the votes and the majority dictates the current game state. This solution minimises delay at the cost of extra bandwidth for both peers and RC. This system also prevents a single RC from attacking peers by sending incorrect results, as they will be discarded in favour of the majority. As with SEA, the RC must be selected securely to prevent collusion. SGA is a possible candidate; however, as with SEA it will add significant delay, undermining the delay benefits of P2P architectures.

2.4 Summary

This chapter covered related work for this thesis. Section 2.1 covered network game properties, including descriptions of four common game genres (FPS, RTS, RPG, and MMOG). Section 2.2 described our cheat model, described all known theoretical and practical cheats, and reviewed previous cheat classifications. Finally, Section 2.3 describes the C/S, MS, and multiple P2P network game architectures; with reference to the properties in Section 2.1. In the following three chapters we propose three different network game architectures using referees to increase their scalability, responsiveness, and fairness.

Chapter 3

The Referee Anti-Cheat Scheme (RACS)

In this chapter we investigate using a referee to increase the scalability, responsiveness, and fairness of the C/S architecture, without increasing the possibilities for cheating. We name the resulting architecture the Referee Anti-Cheat Scheme (RACS). When designing RACS we observed that in traditional sporting events the referee must continuously observe the game, but is only required to intervene when the rules are broken. To mimic this in a network game architecture requires allowing peers to exchange updates directly (P2P), while sending a copy to the referee for validation. Thus, RACS is a hybrid C/S and P2P architecture combining their strengths and overcoming their weaknesses. RACS has the following benefits:

(i) It minimises delay as updates are sent directly between peers.

(ii) It increases fairness, as the peer-to-referee delay is less critical than the client-to-server delay in C/S.

(iii) It provides cheat detection / prevention equal to that in C/S, while reducing delay and the server / referee's outgoing bandwidth and processing requirements.

(iv) It is more effective and efficient than existing P2P cheat solutions [13, 36, 40, 64], as it is also secure against the invalid command, information exposure, and blind opponent cheats with lower cost.

(v) It allows peers with poor connections to continue playing - albeit with higher delay - unlike the protocols in [36, 64].

(vi) Its centralised algorithms calculate the round length more accurately, faster, and with lower bandwidth than the distributed algorithm in [64].

Note that for optimal delay and fairness in both C/S and RACS, the server placement problem must be addressed. However, as this problem is heavily influenced by business considerations [100] it is beyond the scope of this thesis.

This chapter also presents an extended cheat classification that includes all known cheats. Further, it defines the round length adjustment problem, and proposes two solutions. Finally, this chapter describes the method used to construct the realistic simulation inputs used throughout this thesis.

The layout of this chapter is as follows. Section 3.1 presents our cheat classification. Section 3.2 discusses the RACS concept and protocol, including message formats, communication modes, and the message validation procedure. Section 3.3 proposes two possible algorithms for adjusting the round length in RACS. Section 3.4 describes RACS's cheat detection / prevention; presents an analytical analysis of RACS's bandwidth scalability; and uses simulation to evaluate RACS's bandwidth scalability, responsiveness, and the round length adjustment algorithms. Finally, Section 3.5 summarises this chapter. An early version of our cheat classification was published in [139], and revised in [141]. Our original RACS proposal was published in [145], with the round length adjustment published in [140].

3.1 Cheat Classification

We propose classifying cheats into four levels: game, application, protocol, and infrastructure. Game level cheats occur completely within the game program, without any modification or external influence. Application level cheats require either modifying the game executable or data files, or running programs that read from / write to the game's memory while it is running. Developing application level cheats requires knowledge about reverse engineering software; however, using them is trivial. Feng *et al.* [57] provide a detailed list of the techniques used to create application level cheats. Protocol level cheats involve interfering with packets sent and received by the game. Packets may be inserted, destroyed, duplicated, or modified by an attacker. Many of these cheats are dependent on the architecture used by the game (C/S or P2P). Note that the definition of these three levels of cheats are identical to those in [63]. We have renamed network level cheats in reference [63] to infrastructure level cheats, which require modifying or interfering with the software (*e.g.*, display drivers) or hardware (*e.g.*, the network infrastructure) that the game is using. Note that our infrastructure level classification is a super-set of the network level cheats in reference [63] to accommodate recently known cheats (*i.e.*., infrastructure level Information Exposure and Proxies / Reflex Enhancers).

Table 3.1 classifies all known cheats, described in Section 2.2.2, into our four levels. We have added the RMT / Power Levelling, Invalid Command, Undo, and Blind Opponent cheats in the table, and included the Information Exposure and Proxies / Reflex Enhancer cheats in the infrastructure level. Note that the Information Exposure and Proxies/Reflex Enhancers cheats are included in both the Game and Infrastructure levels, depending on how the cheat is accomplished. As shown in Table 3.1, AS [13] and NEO/SEA [36, 64]

Cheat	C/S	CDS	AS	NEO / SEA	P2P-RC	RACS
Game Level						
Bugs	✓	✗	✓	✓	✓	✓
RMT / Power Levelling	✓	✗	✗	✗	✓	✓
Application Level						
Information Exposure, Invalid Command	✓	✗	✗	✗	✓	✓
Bots / Reflex Enhancers	✗	✓	✗	✗	✗	✗
Protocol Level						
Suppressed Update, Timestamp, Fixed Delay, Inconsistency	✓	✗	✓	✓	✓	✓
Collusion	⊠	⊠	⊠	⊠	⊠	⊠
Spoofing, Replay	✓	✗	✗	✓	✓	✓
Undo	–	✗	✓	✗	–	–
Blind Opponent	–	✗	–	✗	–	✓
Infrastructure Level						
Information Exposure	✓	✓	✗	✗	✓	✓
Proxies / Reflex Enhancers	⊠	⊠	⊠	⊠	⊠	⊠

✓ solvable ✗ not yet solved ⊠ not solvable – not applicable

Table 3.1: Game cheats and their possible solutions.

have lower cheat detection / prevention than C/S. Note, CDS are combined with a network game architecture (*e.g.*, C/S or RACS) to detect / prevent cheating. As described in Section 3.4.1, RACS has cheat detection / prevention equal to that in C/S.

3.2 Referee Anti-Cheat Scheme

3.2.1 RACS Concept and Protocol

As shown in Figure 3.1, RACS comprises three entities: an authentication server (S_A), a set of n players $P = \{P_i \mid P_i$ is a player with unique ID $i\}$, and a trusted referee R. A joining player P_i first contacts S_A, which validates P_i (*e.g.*, identity, subscription, banning, *etc.*), and downloads his avatar state to both his host and R. The S_A is also used to store offline players' avatar state, and assigns the unique ID to each player. For these joining steps, we assume the use of existing player-authentication and start-up protocols [4]. A leaving P_i notifies R to upload its avatar state to S_A, which in turn, sends an acknowledgement (ACK) to both P_i and R. Receiving the ACK, P_i disconnects from R.

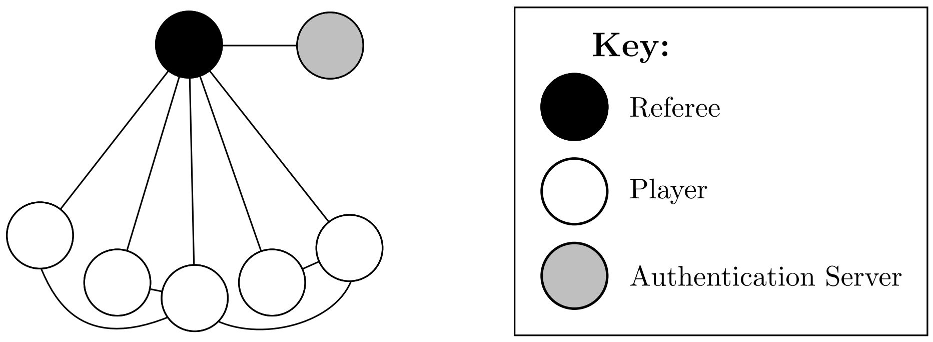


Figure 3.1: RACS architecture.

The referee, R, is a process running on a trusted host, *e.g.*, the server, that has authority over the game state. The referee stores the authoritative game state and simulates / validates player updates to maintain consistency and prevent cheating. For these tasks, it must receive and simulate all player updates. The referee sends updates to peers only if they are unable to communicate directly (in the event of message loss or cheating).

The referee divides game time into rounds of length $\tau \leq \tau_{max}$; the developer sets τ_{max} such that the game is playable. Section 3.3 describes how the referee can reduce τ to increase responsiveness (provided players have good connectivity) or increase τ if players have poor connectivity.

For each round r, every P_i generates a pair $U_i = (r, I)$, to be included in his messages transmitted to R and other peers. Here, I is the information containing P_i's actions (*e.g.*, move, attack, *etc.*) and information about connections with his peers (*e.g.*, informing R about disconnecting from an opponent). The referee initialises the round number $r = 1$. Each copy of r (kept in R and each P_i) is independently incremented for every elapsed time τ. One can use NTP [97] for synchronising rounds between hosts.

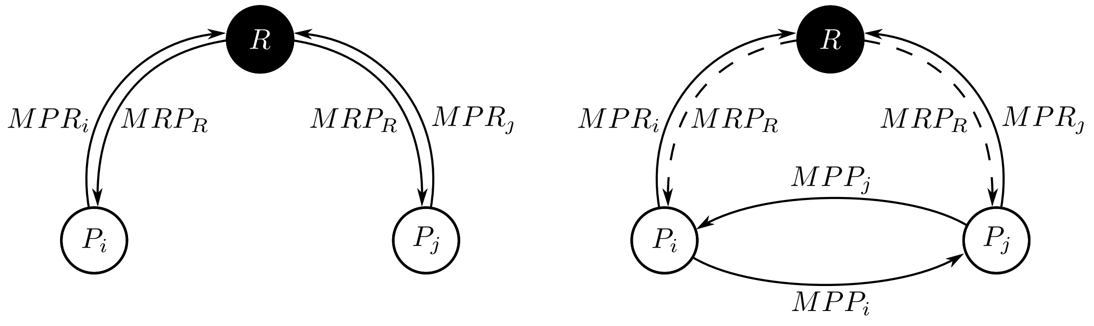
As shown in Figure 3.2, RACS considers three different message formats:

(i) peer to peer message - $MPP_i(U_i)$, signed by the sender P_i;

(ii) peer to referee message - $MPR_i(U_i, S_i, V_i, H_i, T_i)$, signed by the sender P_i;

(iii) referee to peer message - $MRP_R(i, U_i', C_i)$, signed by the sender R;

where S_i is secret information (*e.g.,* health and items); V_i is a bit vector of the MPP_j updates P_i received on-time in the previous round; H_i is the hash $H(U_j)$ for each on-time MPP_j that P_i received in the previous round, combined using exclusive or (XOR); for each late MPP_j received in the previous round, $T_i = \{(j, r_j, H(U_j))\}$; $U_i' = (r', I)$, where r' is the round number of the referee when U_i was received; and C_i is player P_i's digital certificate. As S_i is only included in *MPR* messages, and not *MPP*, S_i is only transmitted to the referee, which is conceptually equivalent to On-Demand Loading [89]. Message components V_i, H_i, and T_i are used to detect the inconsistency cheat (See Section 3.4.1). Player P_i's digital certificate, C_i, is optional and instructs the receiver to begin peer-to-peer communication, discussed in



(a) PRP Mode. (b) PP Mode.

Figure 3.2: RACS communication modes.

Section 3.2.2, with P_i. We assume the referee's digital certificate, C_R, is distributed to all players with the client software. Note that V_i and H_i were added to our earlier description of RACS [145] to reduce the size of T_i.

3.2.2 RACS Communication Modes

As shown in Figure 3.2, the communication between any P_i and P_j that are mutually aware - within each others' Area of Interest (AoI) - can be through the referee R (Peer-Referee-Peer: PRP mode), or direct (Peer-Peer: PP mode). In PRP mode each player sends MPR and receives MRP messages to / from R. This mode provides security equal to that in C/S. In contrast, peers in PP mode exchange MPP messages directly, which reduces delay and R's outgoing bandwidth, while maintaining security. Thus, PP is the preferable mode. Note, in PP mode R sends an MRP (dashed lines in Figure 3.2(b)) only in the event of state inconsistencies due to network delay or cheating.

The referee converts mutually aware PRP peers (*e.g.,* P_i and P_j) into PP mode by sending MRP to P_i and P_j including C_j and C_i respectively. Figure 3.3 shows four interacting peers using a combination of PP and PRP modes. Peer P_i is in PP mode with P_j and P_k, and PRP mode with P_l; P_j is in PP mode with P_i, and PRP mode with P_k and P_l; *etc.* Note that P_i experiences better responsiveness than P_l, with an average delays of $1\frac{1}{3}$ and 2 hops respectively. The lower delay is the incentive to peers to use PP mode, despite the increase in bandwidth.

A peer P_i will revert to PRP mode with another peer P_j if:

(i) P_i's avatar is no longer in P_j's AoI, or vice-versa,

(ii) P_i receives less than p percent of P_j's last $s \geq 1$ messages, or

(iii) P_i does not receive P_j's update for more than $w \geq 0$ consecutive rounds.

Reversion requirement (i) provides AoI filtering to reduce bandwidth; only players that include P_i in their AoI will be updated. Requirement (ii) prevents a cheater repeatedly

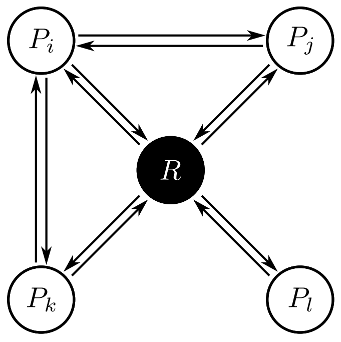


Figure 3.3: Example of mixed PP and PRP communication.

sending one message and then dropping w consecutive messages, while requirement (iii) ensures that losses are not clustered, which would have a large impact on P_i's game-play experience. For either case, P_i sends an MPP to P_j and an MPR to R, that includes I notifying them of the reversion. Then, R only forwards P_i's moves to P_j if P_i is within P_j's AoI, and vice-versa. Requirements (ii) and (iii) form the QoS requirements for PP communication. Honest players with long delays that are unable to meet the PP QoS requirements will be forced to revert to PRP mode. This will disadvantage these players; however, RACS is more tolerant than NEO/SEA which completely remove players unable to communicate within the round. Note that RACS is cheat-proof when $w = 0$ or $p = 100\%$. The optimal values for w, p, and s should minimise PP to PRP reversions, and minimise the number of messages that may be dropped.

In PP mode a player P_i sends his MPP to all PP peers and MPR to R for each elapsed round length τ. Thus, for every round each peer expects a message from all PP peers, and R expects a message from every player. However, due to communication failures or cheating, a message may not arrive. Assume P_j and R are expecting a message from P_i. We consider three cases for lost messages:

(i) Neither receives.

(ii) Only R receives.

(iii) Only P_j receives a message.

In RACS, P_j and R extrapolate (dead-reckon) the avatar of each P_i whose message is not received. The referee's state is authoritative, and it notifies affected players about state inconsistencies caused by extrapolation. In case (i), only P_i may be disadvantaged, as P_j and R have matching state. However, in case (ii), P_j might be slightly disadvantaged if his game state is incorrect. Finally, case (iii) disadvantages both P_i and P_j since R's dead-reckoning may make their states incorrect. Note that for cases (i) and (ii), if the missing message violates reversion requirements (ii) or (iii), P_j will revert to PRP.

In PRP mode P_i sends MPR_i to R for each elapsed τ. In the following round, the referee sends MRP_R including C_i to all P_j that should enter PP communication with P_i, and

MRP_R including the corresponding C_j's to P_i. As in PP mode, the referee and each peer extrapolate P_i's avatar for each missing MPR and MRP, respectively; any inconsistency is resolved using R's authoritative state.

Algorithm 3.1 and Algorithm 3.2 summarise the sequences of steps described in this section for every player and the referee in each round respectively.

Algorithm 3.1: RACS_player_game_loop()

```
   /* The game loop run by each player, $P_i$, in every round. */;
1  begin
2      Read the player input
3      Construct $MPR_i$ and send it to the referee.
4      Construct $MPP_i$ and send it to all PP peers.
5      for every $MPP_j$ received within the round do
6          Use $C_j$ to validate $MPP_j$'s digital signature.
7          Discard $MPP_j$ if validation fails.
8          Use the $r$ in $MPP_j$ to ensure it is newer than all previously received updates.
           Discard all $MPP_j$ except for the most recent.
       end
9      for every $MRP_R$ received within the round do
10         Use $C_R$ to validate $MRP_R$'s digital signature.
11         Use the $r$ in $MRP_R$ to ensure it is newer than all previously received updates.
           Discard all $MRP_R$ except for the most recent.
       end
12     Simulate all $U_j$ in $MRP$ and $MPP$ messages.
13     Use interpolation/extrapolation for missing updates.
14     Display the results to the player.
   end
```

3.3 Round Length Adjustment

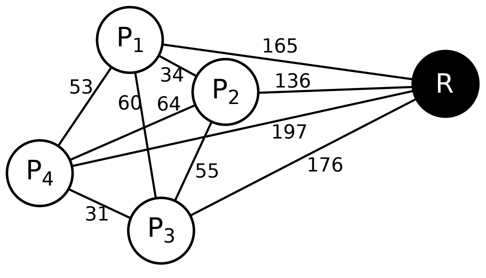
MMOG typically divide the virtual world into discrete regions called zones, and players in different zones cannot interact. Thus, each zone is independent and has its own round length τ. For simplicity, in this section we consider a world consisting of a single zone; however, our results are easily extensible to games comprised of multiple zones.

As each player generates one update per round, reducing τ increases responsiveness. However, reducing τ increases the likelihood of peers failing the QoS requirements and reverting to PRP mode, increasing delay and the referee's bandwidth. The optimal value for τ must balance these constraints.

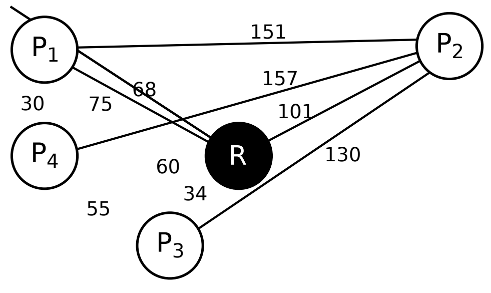
Figure 3.4 shows two example topologies and their corresponding delay matrices. Topology 1 assumes a tight cluster of players with high peer-to-referee delays, such as a group of friends located in one city playing on a server located in a different country. Topology 2 is comprised of several players with low peer-to-peer and peer-to-referee delays, and

Algorithm 3.2: RACS_referee_game_loop()

```
   /* The game loop run by the referee, $R$, in every round. */;
1  begin
2      for every player $P_i$ do
3          if $P_i$ attempted an invalid action (due to inconsistent state) then
4              Send an $MRP_R$ to $P_i$ with the current game state.
           end
5          for every $P_j$ in PRP communication with $P_i$ do
6              if $P_i$ and $P_j$ should enter PP mode. then
7                  Send $MRP_R(j, U_j', C_j)$ to $P_i$.
               else
8                  Send $MRP_R(j, U_j')$ to $P_i$.
               end
           end
       end
9      for every $MPR_i$ received within the round do
10         Use $C_i$ to validate $MPR_i$'s digital signature.
11         Discard $MPR_i$ if validation fails.
12         Compute $H_i'$ by XORing all $H(U_j)$ for every $MPP_j$ $P_i$ received on-time in the
           previous round (indicated by $V_i$).
13         Compute $H(U_j)'$ for every $MPP_j$ received late in the previous round
           (indicated by $T_i$).
14         if $H_i' \neq H_i$ then
15             Request $P_i$ to forward all $MPP_j$ messages used to calculate $H_i$
           end
16         for every $H(U_j)' \neq H(U_j)$ do
17             Request $P_i$ to forward $MPP_j$
           end
18         for every $MPP_j$ forwarded by $P_i$ do
19             Use $C_j$ to validate $MPP_j$'s digital signature.
20             if $MPP_j$'s signature is invalid then
21                 $P_i$ is attempting to frame $P_j$.
22             else if $MPP_j$'s $U_j$ does not match that in $MPR_j$ then
23                 $P_j$ is attempting the inconsistency cheat.
               end
           end
24         Use the $r$ in $MPR_i$ to discard old updates from each player.
       end
25     Simulate all $MPR_i$.
26     Use interpolation/extrapolation for missing updates.
   end
```



(a) Topology 1.



(b) Topology 2.

	P_1	P_2	P_3	P_4	R
P_1	0	34	60	53	165
P_2	34	0	55	64	136
P_3	60	55	0	31	176
P_4	53	64	31	0	197

(c) Topology 1 delay matrix.

	P_1	P_2	P_3	P_4	R
P_1	0	151	75	30	68
P_2	151	0	130	157	101
P_3	75	130	0	55	34
P_4	30	157	55	0	60

(d) Topology 2 delay matrix.

Figure 3.4: Example topologies.

a single player (*i.e.*, P_2) with high delays. This scenario occurs, for example, when the majority of players have DSL/Cable Internet access, but one player is connecting using a dial-up modem. For Topology 1 we believe $\tau = 64ms$ will be optimal, as it will maximise responsiveness for PP updates. Due to the high peer-to-referee delay inconsistencies may occur; however, we believe the increased responsiveness will maximise player enjoyment. For Topology 2, setting $\tau = 157ms$ would frustrate players P_1, P_3, and P_4; instead, setting $\tau = 75ms$ would maximise responsiveness for the majority of players. Player P_2 can still participate, but only in PRP mode. If P_2 wishes to use PP mode he must purchase faster Internet access.

Note, as each game has different delay requirements, the best algorithm for adjusting τ is game specific. In this section we investigate two possible techniques: a brute force approach that minimises the total system delay, and a voting approach with lower processing overhead.

3.3.1 Delay Model and Problem Statement

Let d be a 2-dimensional delay matrix of size $|P| \times |P| + 1$, where $d_{i,R}$ denotes the delay between a player P_i and the referee R, and $d_{i,j}$ denotes the delay between two players P_i and P_j. Note, in this thesis we assume delays are synchronous (*i.e.,* $d_{i,j} = d_{j,i}$) as they are measured using ICMP echo messages (pings), which cannot detect asynchronous delays. Further, let $d_{i,j} = \infty$ if the delay has not been measured between peers P_i and P_j, which may occur if they have not interacted, or are unable to exchange MPP messages.

Since updates are processed at the end of each round, and assuming rounds are pipelined and begin every f *ms*, the Peer-to-Referee Delay for a player P_i, denoted $PRD(i, \tau, f)$, is τ if $d_{i,R} \leq \tau$, or $d_{i,R}$ plus the time until the next round ends. Formally:

$$PRD(i, \tau, f) = \begin{cases} \tau & d_{i,R} \leq \tau \\ f \times \left\lceil \frac{d_{i,R}-\tau}{f} \right\rceil + \tau & d_{i,R} > \tau \end{cases} \tag{3.1}$$

At any point in time each P_i is interacting with a subset of P, denoted $PI(i) \subseteq P$. This subset is further divided into PP peers ($PP(i) \subseteq PI(i)$) and PRP peers ($PRP(i) \subseteq PI(i)$). Note that $PP(i) \cup PRP(i) = PI(i)$, and that $PI(i)$ is typically much smaller than P due to AoI filtering. The Total-Player-Delay for P_i, $TPD(i, \tau, f)$, is the sum of the time taken for an update generated by P_i to reach all P_j, where $P_j \in PI(i)$. Formally:

$$TPD(i, \tau, f) = \tau \times |PP(i)| + \sum_{j \in PRP(i)} [PRD(i, \tau, f) + PRD(j, \tau, f)] \tag{3.2}$$

The Total-System-Delay $TSD(\tau, f)$, is the sum of all total-player-delays and player-referee-delays. Formally:

$$TSD(\tau, f) = \sum_{P_i \in P} [TPD(i, \tau, f) + PRD(i, \tau, f)] \tag{3.3}$$

The round length adjustment problem is to select τ such that Equation (3.3) is minimised. Note that f is typically set by the developer [135].

3.3.2 Round Length Adjustment Algorithms

Unlike NEO [64], which requires a fully distributed round length adjustment algorithm, the referee in RACS is trusted to adjust τ. Using a centralised approach is simpler, faster, and uses less bandwidth, but requires a trusted third party.

3.3.2.1 Brute Force

A simple brute force approach to calculate the optimal τ is to compute $TSD(\tau, f)$ (Equation (3.3)) for $\tau = [1, \tau_{max}]$ and select the minimum. The referee's run-time complexity for the worst case of the brute force approach is $O(\tau_{max}|P|^2)$; therefore, this approach is infeasible if the round length is adjusted frequently and P grows large. The $TSD(\tau, f)$ for $\tau = [0, 200]$ on Topology 2 is shown in Figure 3.5. Note that due to the presence of many local minimums, optimisation techniques such as gradient descent cannot be used.

3.3.2.2 Voting

The brute-force algorithm calculates the minimum total-system-delay; however, it has very high processing overhead and fails to maximise the benefits of the distributed nature of RACS. To reduce the processing requirements we propose a voting approach where each player P_i votes for the τ_i that minimises $TPD(i, \tau_i, f)$ (Equation (3.2)). The referee tallies the votes to construct a Cumulative Density Function (CDF), and selects the minimum τ

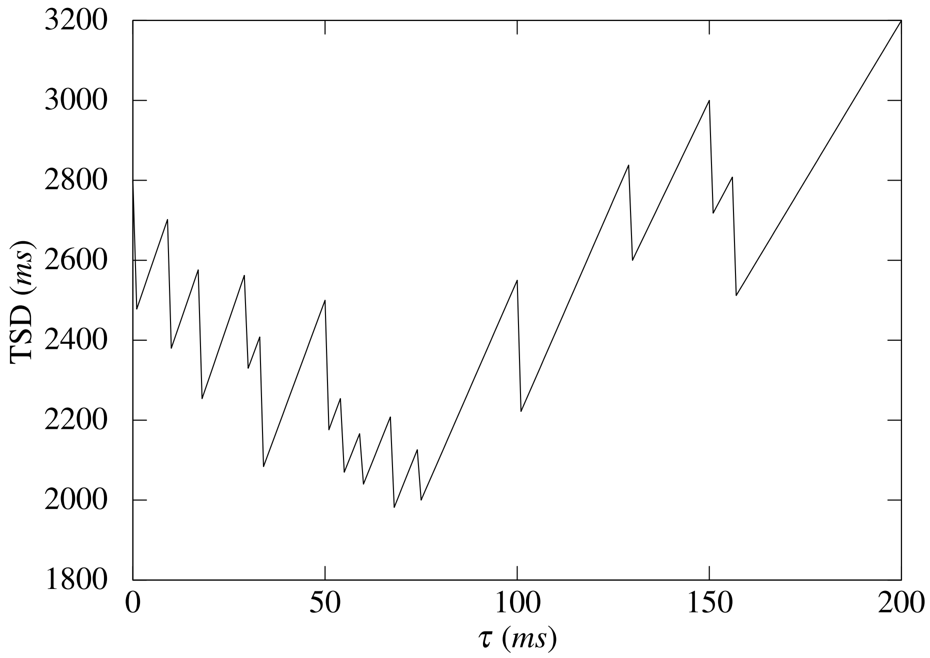


Figure 3.5: TSD for Topology 2.

such that 50% of votes are less than or equal to τ. The peer vote() algorithm is shown in Algorithm 3.3, and the referee tally() algorithm is shown in Algorithm 3.4. As the referee must tally all $|P|$ votes, the worst-case complexity for tally() is $O(|P|)$, far lower than that of the brute force approach. The worst case complexity of vote() is $O(a)$, assuming each player has at most a avatars within his AoI.

Algorithm 3.3: vote(d_i, f)

```
/* $d_i$ - a 1D array of delays between a peer $P_i$ and all $P_j \in PI(i)$ peers and $R$ */
/* $f$ - Round frequency. */
/* Export $\tau_i$ - $P_i$'s vote. */
begin
    $min \leftarrow \infty$
    $\tau_i \leftarrow 0$
    for $j = 1$ to $|d_i|$ do
        if $TPD(d_i,\ d_{i,j},\ f) < min$ then
            $min \leftarrow TPD(d_i,\ d_{i,j},\ f)$
            $\tau_i \leftarrow d_{i,j}$
        end
    end
end
```

3.4 RACS Evaluation

Our evaluation of RACS is divided into four parts: a discussion of how RACS provides cheat prevention equal to that in C/S (Section 3.4.1); an analytical evaluation of the band-

Algorithm 3.4: tally()

```
begin
    Instruct all peers to vote
    Receive $\tau_i$ from all peers
    Construct CDF
    Select the minimum $\tau$ such that 50% of $\tau_i \leq \tau$
    Notify all peers of $\tau$
end
```

Cheats Prevented	**RACS Strategy**
Information Exposure	On-demand Loading
Invalid Command	Referee simulation and validation
Suppressed Update	Referee authoritative state
Replay	Round number r
Spoofing	Signed messages
Undo	No commit / reveal steps
Timestamp	Timestamps not used for event ordering
Cheats Detected	**RACS Strategy**
Bugs	Software patches
RMT / Power Levelling	Statistical analysis of log files
Bots / Reflex Enhancers	Combine RACS with a CDS
Fixed Delay	QoS requirements and reversion to PRP
Inconsistency	Signed messages and comparing hashes
Blind Opponent	Reversion to PRP mode
Undetectable Cheats	**RACS Strategy**
Collusion	None
Proxies / Reflex Enhancers	None

Table 3.2: Summary of RACS cheat detection / prevention strategies.

width requirements of RACS (Section 3.4.2); three simulations evaluating the delay and bandwidth of RACS (Section 3.4.3); and one simulation evaluating the round length adjustment algorithms (Section 3.4.4).

3.4.1 Cheat Prevention

RACS in PRP mode provides security equivalent to that in C/S as it uses a trusted entity to simulate the game and forward updates. Obviously cheat solutions in PRP are similar to those in C/S and, thus, are not discussed. In the following, we explain how RACS in PP mode addresses various cheats. Table 3.2 summarises the methods used by RACS to detect and prevent cheating. Throughout this discussion, we assume a referee R, a cheating P_{C}, and P_{C}'s opponent P_{H}.

Bugs: RACS assumes that bugs will be fixed by software patches (released by the publisher), as does C/S.

RMT / Power Levelling: As the referee receives a copy of all updates it can perform statistical analysis of its log files to detect RMT / Power Levelling, identical to C/S.

Information Exposure: As P_C does not receive P_H's secret information (S_H), information exposure is impossible. S_H is sent only in MPR; see Section 3.2.1. This is equivalent to On-Demand Loading [89].

Invalid Command: The referee simulates / validates all player updates to prevent invalid commands. Further, the authentication server (S_A) stores all offline players' avatar state to prevent tampering.

Bots / Reflex Enhancers: As in C/S [47], one may combine RACS with a Cheat Detection Scheme (CDS) such as PunkBuster or VAC to detect bots / reflex enhancers.

Suppressed Update: In RACS a cheater P_C may suppress updates to the referee and/or peers. As the referee's state is authoritative, and as it extrapolates avatar movement when updates do not arrive, suppressing updates to the referee will only disadvantage the cheater. The QoS requirements between peers for PP communication, discussed in Section 4.1.2, ensure that a cheater cannot suppress a significant number of updates; else, the victim will revert to PRP communication, preventing this cheat. See Section 4.1.2 for more details.

Timestamp: The round numbers in MPR / MPP messages are not used for event ordering. Only round numbers in MRP messages are used for event ordering. Hence, this cheat is prevented.

Fixed Delay: Applying fixed delay may make P_C violate reversion requirements (ii) or (iii) (see Section 3.2.2) causing P_H to revert to PRP mode with P_C. This will punish P_C because his delay (with respect to P_H) will be two hops, while that of P_H (with respect to the other PP peers) will be one hop (See Figure 3.3). If P_C's message arrives within the round (not late), RACS does not consider it cheating. Since late updates are indistinguishable from cheating, this solution may penalise honest but slow players; nevertheless, it is better than [36, 64] which prohibit slow peers from playing.

If the total delay from P_H to P_C and P_C to R is below τ it is possible for P_C to delay sending his MPR until after he has received P_H's MPP. However, as P_C's update must be received within the same round to gain an advantage, and assuming S_A adjusts τ corresponding to the player delays, this is unlikely to occur and would provide almost no advantage.

Inconsistency: The V_i, H_i, and T_i, components of MPR messages are used to detect the inconsistency cheat following Algorithms 3.1 and 3.2. When an inconsistent hash is detected the referee must determine if it is the inconsistency cheat, or if a cheater is attempting to frame another peer. The following two cases describe the sequence of steps for either eventuality; see Section 3.2.1 for the message formats.

Case 1: P_C sends a different U_C to P_H and R (inconsistency cheat):

1. When P_H receives $\mathrm{MPP_C}$ it validates the digital signature against C_C to ensure the message is not spoofed or corrupt (Algorithm 3.1 lines 6 and 7).
2. Assuming $\mathrm{MPP_C}$ arrived on-time the hash $H(U_C)$ is XOR'ed into H_H and sent to R in the following round (Algorithm 3.1 line 3).
3. Referee R validates the digital signature of $\mathrm{MPR_H}$ to ensure it is not spoofed or corrupt (Algorithm 3.2 lines 10 and 11).
4. The referee calculates H'_H from the U_i in each MPR_i in V_H, including the conflicting U_C (Algorithm 3.2 line 12).
5. As $H_H \neq H'_H$ the referee requests P_H to forward all MPP_i used to construct H_H (Algorithm 3.2 lines 14 and 15).
6. After receiving the reply from P_H and validating its digital signature, the referee calculates $H(U_i)$ for each received MPP_i to determine that MPP_C is the conflicting update.
7. The referee validates the digital signature of $\mathrm{MPP_C}$ against C_C to verify the cheat.

If $\mathrm{MPP_C}$ arrives late in Step 2, $H(U_C)$ is included in T_H rather than H_H. Further, steps 4-6 are simplified as the referee validates each hash individually; thus, it will request P_H to forward $\mathrm{MPP_C}$ only.

Case 2: P_C attempts to frame P_H:

1. P_C constructs $\mathrm{MPR_C}$ with an invalid H_C and sends it to R (Algorithm 3.1 line 3).
2. Referee R validates the digital signature of $\mathrm{MPR_C}$ to ensure it is not spoofed or corrupt (Algorithm 3.2 lines 10 and 11).
3. The referee calculates H'_C from the U_i in each MPR_i in V_C (Algorithm 3.2 line 12).
4. As $H_C \neq H'_C$ the referee requests P_C to forward all MPP_i messages used to create H_C (Algorithm 3.2 lines 14 and 15).
5. Cheater P_C sends the required MPP_i messages to R, including an invalid MPP_H.
6. After receiving the reply from P_C and validating its digital signature, the referee calculates $H(U_i)$ for each received MPP_i to determine that MPP'_H is the conflicting update.

7. The referee attempts to validate the digital signature of MPP_H, but determines that it is invalid. As P_C would not accept an MPP with an invalid digital signature, and as all messages from P_C to R were digitally signed to prevent spoofing and corruption, P_C must be attempting to frame P_H.

In Step 1, if P_C attempts to frame P_H with an invalid hash in T_C, steps 3, 4, and 6 are simplified as the referee validates each hash individually; thus, it will request P_C to forward MPP_H only.

Note that each peer must store received MPP messages so that they can be forwarded to R in the event of an inconsistency. The messages are retained for a publisher defined number of rounds before being discarded. The retention length must be long enough for the referee to detect and request inconsistent messages; however, increasing the retention length increases the memory required by peers to store received MPP messages.

Collusion: RACS does not address the collusion cheat, nor do existing P2P and C/S solutions [150].

Spoofing: RACS prevents spoofing by authenticating the signature of every message received. As digital signatures cannot be forged it is impossible for a player to send an update masquerading as another player.

Replay: A message is invalid and is discarded if another message with equal or greater round r has been received (see Section 4.1.2); hence, r is a nonce to detect the replay of old messages.

Undo: RACS does not use the commit / reveal steps as in [36, 64], and thus the undo cheat is impossible.

Blind Opponent: As dropping updates causes P_C to violate the QoS requirements, P_H and P_C will revert to PRP communication, preventing this cheat.

Proxies / Reflex Enhancers: As with existing P2P and C/S solutions [111], RACS cannot prevent proxies / reflex enhancers.

3.4.2 Bandwidth Analytical Analysis

To analyse the bandwidth requirements of RACS requires knowing the sizes of MPP, MRP, and MPR messages. Let $|x|$ denote the size of a message/component x, and let o be the overhead for each message (digital signature, packet header, *etc.*) Note, that $|o|$ is equal for all message types. Several studies [31, 55, 83, 127] have shown the mean and standard deviation of $|U_i|$ is small; thus, overhead o will account for a significant portion of the MPP

		C/S & RACS (PRP Mode)	**RACS (PP Mode)**
Client/Peer	**In**	$o + a \times \|MRP\|$	$a \times (o + \|MPP\|)$
	Out	$o + \|MPR\|$	$o + \|MPR\| + a \times (o + \|MPP\|)$
Server/Referee	**In**	$n \times (o + \|MPR\|)$	$n \times (o + \|MPR\|)$
	Out	$n \times (o + a \times \|MRP\|)$	*negligible*

Table 3.3: Bandwidth comparison of C/S and RACS.

message size, |MPP|. For each round player P_i requires U_j about each avatar within his AoI. To reduce the bandwidth consumed by message overheads, we assume each MRP sent to peer P_i aggregates all relevant U_j. Thus, the MRP message size, |MRP|, grows linearly with the number of PRP peers. However, MRP does not increase in size if peers use PP communication as U_j is exchanged using MPP messages. Unlike |MRP|, the size of MPR messages, |MPR|, increases in PP mode. When all peers use PRP mode - equivalent to C/S - |MPR| has low mean and standard deviation [31, 55, 83, 127]. As more peers use PP mode V_i and T_i expand, increasing |MPR|. However, provided the majority of updates arrive on-time, T_i remains small and each additional PP peer requires only 1 bit in V_i. As the PP mode QoS requirements prevent many late updates, T_i does not significantly increase |MPR|. Note, the size of H_i is fixed regardless of the number of PP peers. Combining all components, while |MPR| for peers using PP mode will be larger than client-to-server messages in C/S, we anticipate their sizes will be similar.

Table 3.3 shows the bandwidth requirements for C/S and RACS assuming there are a avatars within each player's AoI. Note that RACS with all peers using PRP mode (worst case) is equal to C/S. It is obvious that the peers' incoming and outgoing bandwidth increase in PP mode (best case), as they assume responsibility for broadcasting their state updates to other players. Likewise, when all peers use PP mode the referee's outgoing bandwidth is negligible. Further, as peer updates are not aggregated into one message, message overheads consume additional peer bandwidth. Despite the increase in bandwidth PP mode has lower game delay, which is important for network games (see Section 2.1.3), providing incentive for peers to use PP mode. While the server's outgoing bandwidth in PP mode is negligible, it must still receive all MPR messages. As |MPR| grows according to the number of PP peers (albeit very slowly) the referee's incoming bandwidth will increase slightly faster than linearly, possibly causing a bottleneck. Using multiple referees (Chapters 4 and 5) is one possible solution for this problem. Note that the referee's outgoing bandwidth in Table 3.3 does not include the cost for initiating PP communication (two MRP messages). As network games generate hundreds of updates per minute [31, 55, 83, 127], and as commercial MMOG require players to undertake long quests together (minimising PRP/PP transitions), this will not significantly increase the referee's bandwidth requirements.

Note, NEO/SEA [36, 64], an example of anti-cheat P2P game architecture, requires more bandwidth than RACS. Without a trusted third party to compare hashes, messages sent between peers in NEO/SEA must include the hash of all received updates. Thus, peers' incoming and outgoing bandwidth are $O(a^2)$; much larger than the |MPP| in RACS, significantly limiting the maximum group size.

3.4.3 RACS Bandwidth and Delay Simulations

To evaluate RACS we developed the *Network Game Simulator* (NGS) [138], and used it to run three simulations evaluating the different aspects of RACS. Simulation 3.1 evaluates the bandwidth scalability and responsiveness of RACS against C/S in a cheat-free environment. Simulation 3.2 measures the impact of cheating on responsiveness and the referee's bandwidth. Finally, Simulation 3.3 demonstrates the effect of selecting w, s, and p values with packet loss and cheating.

To generate meaningful simulation results requires using appropriate inputs, *e.g.,* the network topology, player distributions, player session lengths, *etc*. To achieve this we constructed two classes of simulation inputs, artificial and realistic, for the three simulations. The artificial inputs are basic models designed to produce simple results that are easy to understand and clearly demonstrate system behaviour. The artificial topologies often, but not always, demonstrate the best case scenario, and avoid the server placement problem which is not addressed in this thesis (See the introduction to Chapter 3). Note, the artificial inputs are specific for each simulation, and are described in each corresponding section. To generate realistic simulation results requires realistic inputs. Unfortunately, there are currently no publicly available traces of MMOG, and very little publicly available information. Further, to model a P2P or hybrid architecture requires knowing the peer-to-peer delays, which are not measured by C/S games, increasing the difficulty of generating a realistic topology. To solve this problem we combined real-world data sets from a popular *Counter-Strike* server [55] (*mshmro.com*), the *hostip.info* service [68], the *Dimes project* [121], and the *PingER Project* [93] to generate realistic inputs. The following section describes how we generated the realistic inputs. We have used these inputs for our simulations in this chapter and also in chapters 4, and 5.

3.4.3.1 Generating Realistic Inputs

Mshmro.com runs a *Counter-Strike* server located in Beaverton, Oregon, allowing up to 22 players to participate simultaneously in this popular FPS [56]. We used the March 2007 server log file which includes the IP addresses of all 11525 players. Using the *hostip.info* service [68] we successfully resolved the cities of 7217 out of 11525 players (the 4308 unresolved addresses were discarded).

As the delay between hosts is not measured by *Counter-Strike*, we utilised the March 2007 CityEdges Internet trace to calculate the player to player delays; Shavitt and Shir [121] described how the trace was generated. The CityEdges file contains the longitude and latitude of 17294 cities (including the 7217 player's cities) and the links between them. We calculated the delay of each link by estimating its length (in *km*) using the Haversine formulae and dividing it by the speed of light in optic fibre. Note that 55 of the 17294 cities were unreachable and were removed. Dijkstra's single source shortest path algorithm was used to construct a city delay matrix, in which each matrix element stores the delay between two cities. To construct our realistic topology the *mshmro.com* players were combined with the city delay matrix and given a 20*ms* last hop delay, corresponding to the delay suffered by broadband hosts [86]. By using Dijkstra's algorithm, the resulting matrix does not model the routing policies used; hence, there are no Triangle Inequality Violations (TIV) which are present and persistent in the Internet [157], and delays are synchronous. To simulate non-optimal routing policies we randomly selected 2.0×10^6 matrix elements and tripled their delay to introduce 1.07×10^6 TIV (17.12%), which is comparable to measurements made of the Internet [157]. Further, our model assumes the delay between two hosts is the maximum in either direction.

In addition to the artificial and realistic topology, it is useful to model groups of interacting players. In particular, previous studies have shown that:

- Interacting players are often located in the same area of the network [33].
- Player sessions are diurnal [10, 55, 110].
- Players prefer servers with low delay [10, 55].

We address these behaviours in our simulation inputs as follows. As the maximum capacity of the *mshmro.com* server is 22 players; and, as the virtual world in *Counter-Strike* is small compared to MMOG, we assumed all participating players are interacting. We generated 100 sets of interacting players from the *mshmro.com* server log by extracting the IP address of players that kill an opponent or are killed in a random 10 minute interval.

3.4.3.2 Simulation 3.1: RACS vs C/S in a Cheat-Free Environment

To increase scalability most MMOG divide the virtual world into regions, each with a fixed maximum capacity. Players cannot enter a region at full capacity, and players in different regions cannot interact. Therefore, the bandwidth and processing requirements grow linearly with the number of regions. We simulate a world comprising a single region with size 1000×1000 units. All players were honest, and we varied the population from 100 to 500 concurrent players, each generating 20 updates per second (a new update every 50*ms*). Note that *WoW* regions support up to 300 players [110]. As the world size is fixed the number of player interactions increases faster than linearly relative to the population size. Each avatar has an AoI radius of 100 units, and its movement is controlled using the Random-Way-Point (RWP) mobility model [75], with a velocity of two units/s and a wait time of zero.

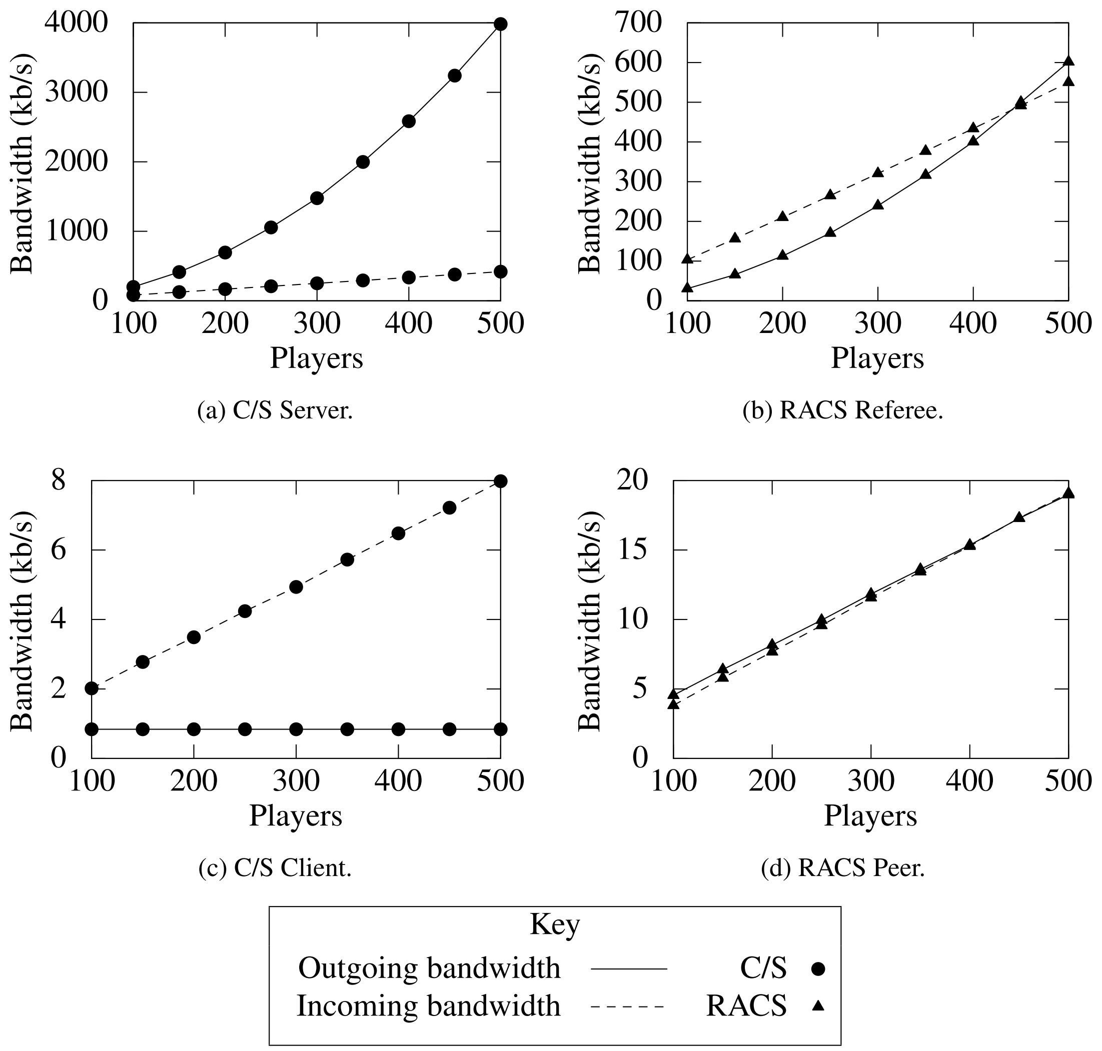
For an even comparison of RACS and C/S bandwidth requirements, the C/S architecture used the MPR and MRP message formats, and the server and referee both aggregate updates. For RACS, $\tau = 150ms$ and $w = 0$ (thus, p and s were irrelevant). Note that RWP is the worst-case scenario for RACS as players frequently moved in-and-out of PP communication, requiring the referee to send updates. In practise the referee's outgoing bandwidth will be lower as commercial MMOG require groups of players to undertake long quests together, minimising the PRP/PP transitions.

We ran the simulation using an artificial topology and the realistic topology. For the artificial topology we measure all communications in hops; thus, the server/referee is placed optimally (*i.e.*, 1 hop from all players). Note that non-optimal placement would significantly impact C/S as all updates must be routed through the distant server, dramatically increasing delay. The impact would be lower on RACS peers using PP mode; however, a low peer-to-referee delay is still desirable as the referee's state is authoritative. For the realistic topology we selected a host in Beaverton, Oregon - the *Counter-Strike* server's location [56] - to act as the server/referee, and the players were randomly selected from the remaining hosts. We removed the 20ms last hop delay from the server/referee as we assume it is well provisioned with a low-latency link.

The incoming and outgoing bandwidth for C/S and RACS using the realistic topology are shown in Figure 3.6. The artificial topology results are comparable. As shown in Figure 3.6(a), the server's outgoing bandwidth in C/S increases faster than linearly, creating a bottleneck. Figure 3.6(b) shows the referee's outgoing bandwidth growing slightly faster than linearly; however, even with 500 players it is only marginally greater than the incoming bandwidth. The faster than linear increase is due to the large number of PRP to PP transitions caused by the mobility model. In practise we expect the referee's outgoing bandwidth to be lower than its incoming. It is obvious from the figures that RACS uses far less bandwidth than C/S.

Figures 3.6(c) and 3.6(d) show that clients' and peers' incoming bandwidth scale linearly. However, as each peer must broadcast updates to its PP peers, their outgoing bandwidth also scales linearly, unlike the fixed outgoing bandwidth of C/S clients. Due to the massive success of P2P networks [126], we believe the increase in peer bandwidth will not cause a bottleneck. Further, the lower delay of PP mode will motivate players to use direct communication.

Figure 3.7 shows the average delay of game state updates for clients/peers in Simulation 3.1; *i.e.*, the time between a player generating a game state update and it being received by another player. As the majority of updates in RACS are sent directly, the average delay for the artificial topology (Figure 3.7(a)) is approximately 1.1 hops, compared to 2 hops for C/S. For the realistic topology (Figure 3.7(b)) the RACS delay is only $9ms$ (12%) faster on average. As the player trace is extracted from a single *Counter-Strike* server, and as *Counter-Strike* players gravitate to servers with low delay [55], it is expected that this simulation shows only a 12% improvement over C/S. In practise, we expect the delay dif-



(a) C/S Server.

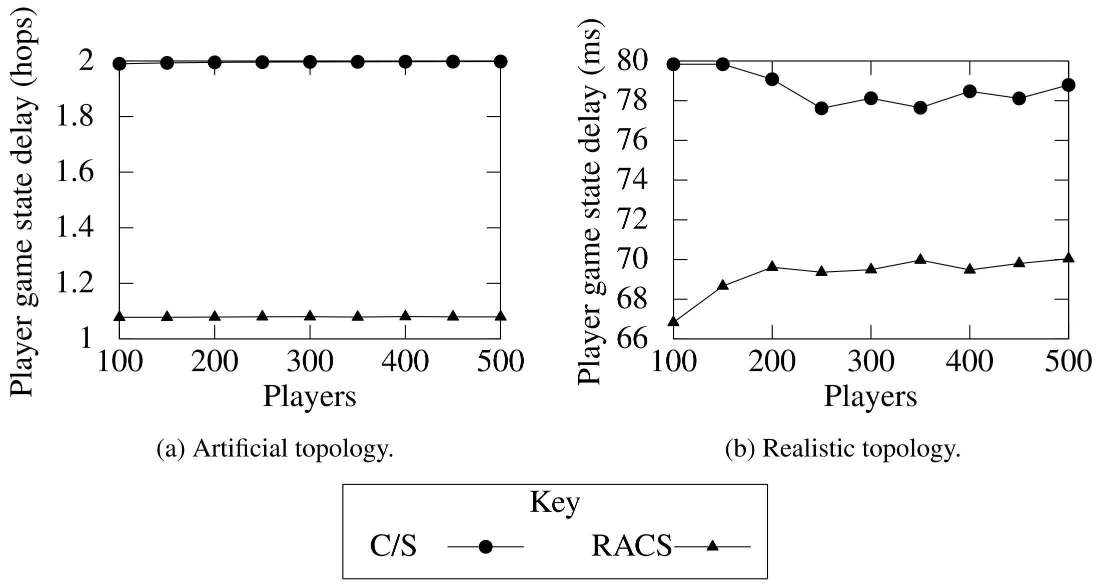
(b) RACS Referee.

(c) C/S Client.

(d) RACS Peer.

Figure 3.6: C/S and RACS bandwidth scalability with increasing players.

ference between RACS and C/S to be considerably larger, as RACS will allow players in distant parts of the network to participate. Note, the RACS results only represent the communication delay and do not incorporate the round length. Therefore, these results assume the round length is optimal, as updates are only processed at the end of the round.



(a) Artificial topology.

(b) Realistic topology.

Figure 3.7: C/S and RACS game state delay.

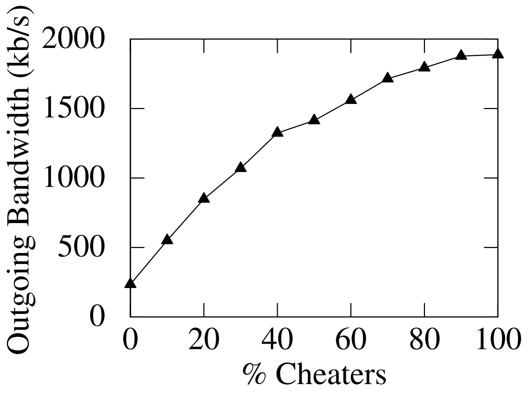
3.4.3.3 Simulation 3.2: RACS Bandwidth and Delay with Cheaters

Simulation 3.2 evaluates the impact of cheaters on the referee's bandwidth and the peers' game state update delay. The player population is fixed at 300 - the maximum region population in *WoW* [110] - and we vary the percentage of cheating peers, c, between 0% and 100%. A cheating peer does not send MPP messages, as in the *blind opponent* and *suppressed update* cheats. Suppressing MPP messages will result in a QoS violation and the peers revert to PRP mode. Two peers do not attempt to re-establish PP communication for at least 60 seconds. All other parameters are identical to Simulation 3.1.

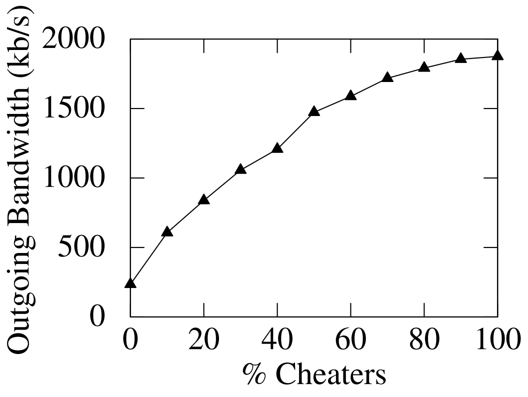
Figure 3.8 shows the referee's outgoing bandwidth, and the delay in updating peers' game state, for the artificial and realistic topologies. As expected, the figure shows that the best case occurs when c=0% as all updates are exchanged directly using PP mode; and the worst case occurs when c=100% as all updates are routed through the referee using PRP mode. The figure shows that RACS scales well, even in the presence of cheaters, as honest players continue to exchange updates. Further, the increasing delay will be a deterrent to cheating.

3.4.3.4 Simulation 3.3: The Effects of w, s, and p, with Packet Loss and Cheating

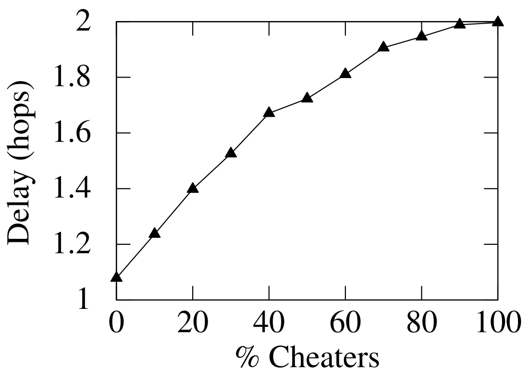
We consider the *Source Engine* used in *Counter-Strike* [135] to demonstrate how w, s, and p, are determined. As *Counter-Strike* is an FPS, it requires low delay [40]; hence,



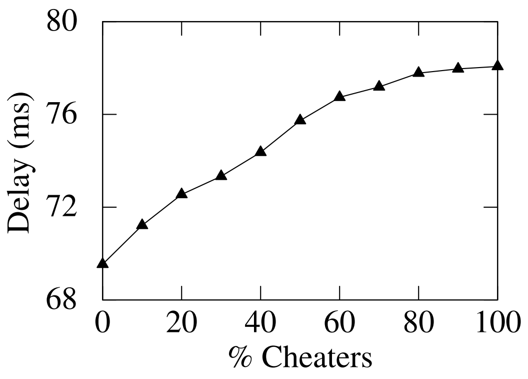
(a) Artificial topology bandwidth.



(b) Realistic topology bandwidth.



(c) Artificial topology delay.



(d) Realistic topology delay.

Figure 3.8: RACS bandwidth and delay with increasing cheaters.

this illustration is applicable to all genres of games, including MMOG. The *Source Engine* generates one update every 50 milliseconds (20 updates per second). As most clients' delay exceeds this, messages are pipelined. The client postpones rendering received updates by $100ms$, and uses interpolation to smooth player transitions (see Section 2.1.3). In the event of two consecutive lost messages, the *Source Engine* client software extrapolates avatars for up to $250ms$; therefore, the client stops rendering the game after seven consecutive losses. Note that Dick *et al.* [48] found that $150ms$ delay did not significantly affect *Counter-Strike* players.

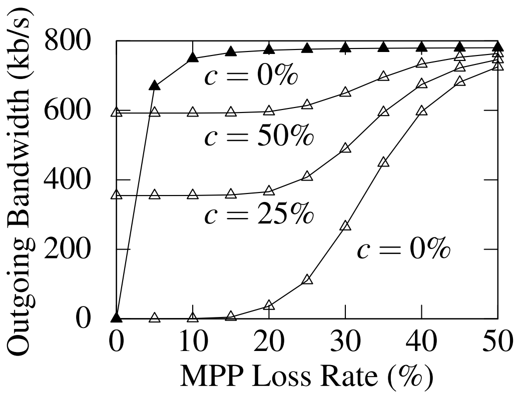
From the described specifications $\tau = 150ms$, and a new round begins every $50ms$. As the engine stops rendering after 7 consecutive lost updates we set $w = 6$. We estimate that losing $2 \times w = 12$ messages per 10 seconds will give a cheater an insignificant advantage. Thus, $s = 200$ (*i.,e.*, 10 seconds / $50ms = 200$), and $p = 94\%$ (*i.e.*, $(1 - \frac{12}{200}) \times 100\% = 94\%$). As communication between peers and the referee does not influence PP communication the simulation does not include MRP or MPR losses. The simulation varies MPP loss rates from 0% to 50%. Also, all messages arrive either on-time or not at all (no late messages). Since the effects of modifying w and p can only be observed when players have repeated interactions, we simulated 40 players - the maximum group size in *WoW* [148] - each with an AoI radius of 200, in a world of size 100×100 units; hence, all players are constantly mutually aware. We simulated the *Source Engine* with 0 ($c = 0\%$), 10 ($c = 25\%$), and 20 ($c = 50\%$) cheaters. All other parameters are identical to Simulation 3.1.

Figure 3.9 shows the referee's bandwidth and the delay in updating peers' game state for the artificial and realistic topologies. The figure also includes a worst-case base line, *i.e.*, $w = 0$, $p = 100\%$, $c = 0\%$. With no cheaters, it is obvious that increasing w and reducing p greatly reduce the referee's outgoing bandwidth and the players' game delay (highest vs. lowest plots in the figure).

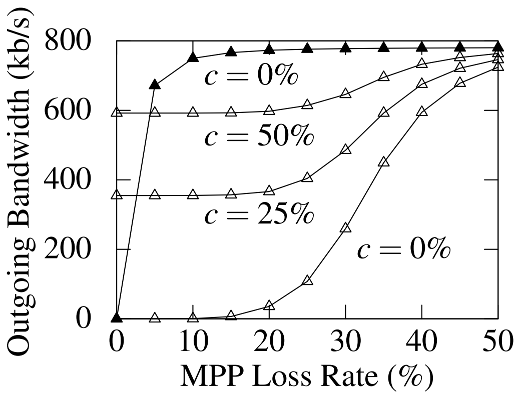
The figure also shows that with $w = 6$ and $p = 94\%$ RACS is highly tolerant to loss. Irrespective of the number of cheaters, loss rates below 15% do not impact the outgoing bandwidth and the average delay; beyond 15%, RACS performance degrades rapidly. However, the results show that increasing numbers of cheaters has a greater impact on performance than loss rate; as more peers revert to PRP mode. Nevertheless, the upper bound of RACS delay is two hops (as in C/S), below NEO/SEA's three-hop bound [36, 64]. In addition, as discussed in Section 3.4.2, its overall bandwidth never exceeds that in C/S and NEO/SEA.

3.4.4 Round Length Adjustment Simulation

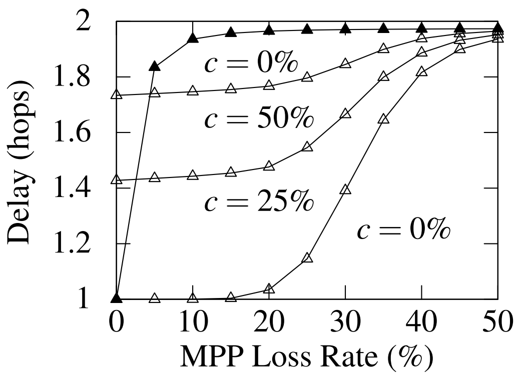
To evaluate the performances of the brute force and voting algorithms in Section 3.3.2, we applied both to the 100 sets of interacting players on the realistic topology (see Section



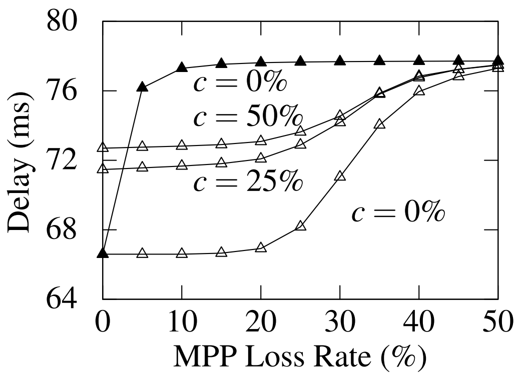
(a) Artificial topology bandwidth.



(b) Realistic topology bandwidth.



(c) Artificial topology delay.



(d) Realistic topology delay.

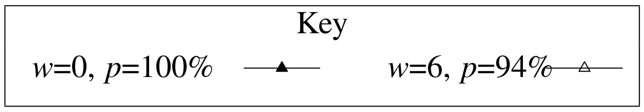


Figure 3.9: RACS with increasing message loss and cheaters.

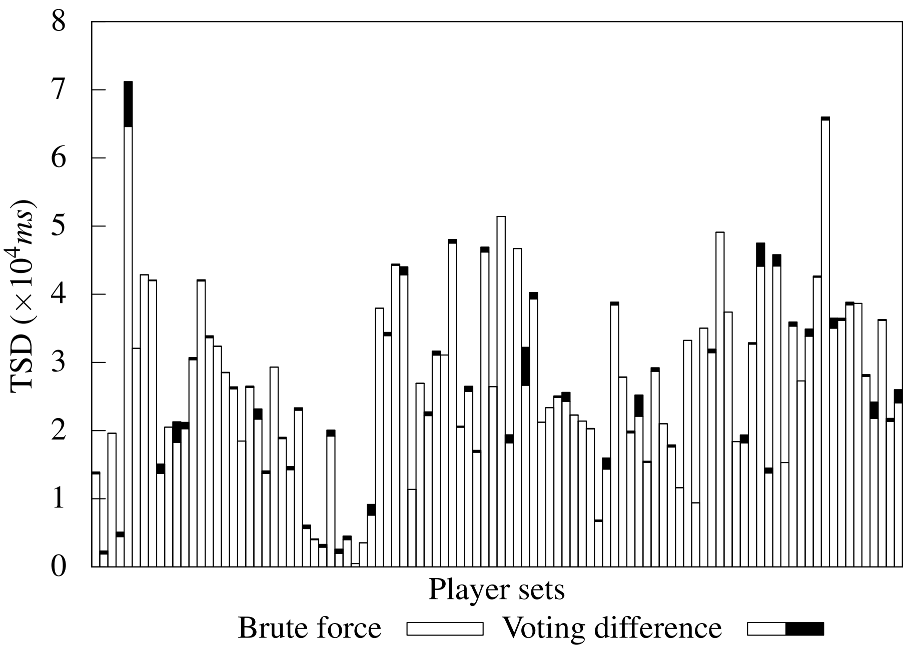


Figure 3.10: Total System Delay.

3.4.3.1). Figure 3.10 shows the Total-System-Delay (TSD) for both algorithms. The results show that, for the majority of player sets, the voting algorithm is close to optimal (Figure 3.10 shows the difference in black). Further, the range and standard deviation of τ for the voting algorithm is lower than those of the brute force algorithm ($41ms$ vs $71ms$ and $7.18ms$ vs $8.98ms$ respectively). As players can modify their playing style to compensate for high delay, but struggle when it fluctuates [14, 15], this suggests players will perform better when using the voting algorithm, even though the TSD is not optimal.

3.5 Summary

In this chapter, we modified and expanded the cheat classification in [64] to incorporate all known theoretical and practical cheats. We proposed the Referee Anti-Cheat Scheme (RACS) - a hybrid C/S and P2P architecture - and showed that by using a referee as a trusted 3rd party RACS provides cheat protection equal to that in C/S. In addition, by allowing direct communication between players RACS has better scalability, responsiveness, and fairness than C/S. Further, we propose two centralised round length adjustment algorithms - an optimal brute force approach, and a faster heuristic approach - that are both faster, more accurate, and require less bandwidth than the distributed algorithm proposed in [64]. We used analytical analysis to show that RACS has better bandwidth scalability than C/S, and used simulations to confirm this. The simulations also show that direct communication between players gives RACS lower delay than C/S; 1.1 hops vs 2 hops for the artificial topology, and a 12% improvement for the realistic topology. Further, Simulation 3.3 shows the impact of the PP mode QoS parameters (w, s, and p) when packet loss and cheating occur. Finally, the analytical analysis of our round length adjustment algorithms showed that our voting algorithm is significantly faster than our brute force approach, and using simulation we showed it produces nearly optimal results. In Chapters 4 and 5 we investigate using multiple referees to further increase RACS scalability and responsiveness.

Chapter 4

The Mirrored Referee Anti-Cheat Scheme (MRACS)

In Chapter 3 we proposed RACS, which uses a referee in the server to improve the scalability, responsiveness, and fairness of C/S, without enabling cheating. As with current C/S MMOG [23, 85], to support thousands of players the referee in RACS must consist of many hosts, co-located in a data centre. Using multiple hosts improves the processing scalability of the referee. However, with all hosts located in one data centre, several problems still exist:

1. The referee's incoming bandwidth may create a bottleneck as it is provisioned at a single location.

2. By allowing PP communication, the player-to-referee delay is less critical in RACS than C/S; however, players with low delay still have an unfair advantage as the referee's game state is authoritative.

3. The referee is a single point of failure.

Note that these problems are also applicable to the C/S architecture, and that problems 1 and 2 are far more severe in C/S than in RACS. The Mirrored Server (MS) architecture, discussed in Section 2.3.2, improves C/S to address these problems. However, player updates in MS, like in C/S, are still routed through mirrors; increasing delay, reducing fairness, and consuming the mirrors' bandwidth and processing power. This chapter investigates combining the RACS protocol with mirrored referees to improve both RACS and MS. We call the resulting architecture the Mirrored Referee Anti-Cheat Scheme (MRACS).

The benefits of MRACS over RACS are:

1. The referee bandwidth bottleneck is reduced as it can be provisioned at multiple locations throughout the network.

2. The player-to-referee delay is reduced - increasing responsiveness and fairness - as each player connects to a close mirror.

3. The single point of failure is removed. While mirror crashes will still occur, only the players connected to the crashed mirror will be affected, and they can re-connect to a different mirror to continue playing.

The benefits of MRACS over MS are:

1. It reduces the mirrors' outgoing bandwidth and processing requirements, without increasing the risk of cheating.

2. It minimises delay as updates are sent directly between players.

3. It increases fairness, as the player-to-mirror delay is less critical than in MS.

To maximise game responsiveness in MS, players connect to their closest mirror, reducing the delay compared to C/S. For this scheme, the publisher must provision every mirror to support its maximum resource usage. As player sessions are diurnal [55, 110], in both MS and MRACS each mirrors' resource usage will vary significantly throughout the day, and thus each mirror will be underutilised most of the time. To reduce the required capacity at each mirror, we propose using a Client-to-Mirror Assignment (CMA) algorithm that assigns players to mirrors such that the total delay of all players is minimised. This maximises responsiveness while preventing saturated or underutilised mirrors. We propose two pairs of algorithms, the optimal J-SA/L-SA and the faster heuristic J-Greedy/L-Greedy. Our solutions are also applicable to other applications with mirrored resources and long-term connections (*e.g.,* Content Distribution Networks [78] or video streaming sites such as *Hulu* [72]). Note that for optimal results the mirror placement problem must also be addressed [117]. However, as mirror placement is heavily influenced by business considerations [100], this problem is beyond the scope of our work.

The layout of this chapter is as follows. Section 4.1 proposes MRACS, including its protocols, communication modes, mirror synchronisation, and security. Section 4.2 investigates the CMA problem and proposes two possible solutions. Section 4.3 uses analytical analysis and simulation to evaluate MRACS and our proposed CMA algorithms. Finally, Section 4.4 summarises the chapter. The MRACS concept and CMA algorithms were previously published in [144] and [142] respectively.

4.1 Mirrored Referee Anti-Cheat Scheme

4.1.1 MRACS Concept and Protocol

Figure 4.1 shows the MRACS architecture. Each mirror hosts a RACS-like referee, and players interact in the PP or PRP modes of RACS (See Section 3.2.2). Since each trusted / authoritative mirror simulates the game and stores the current state, MRACS maintains game consistency and counters cheating.

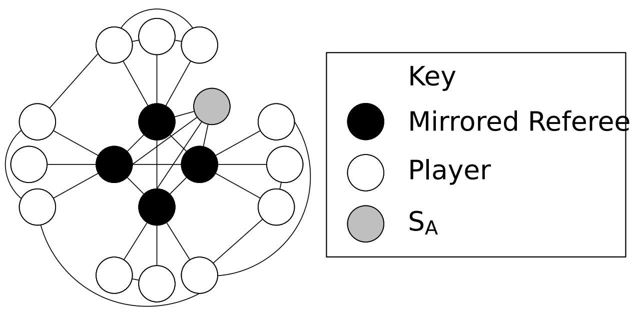
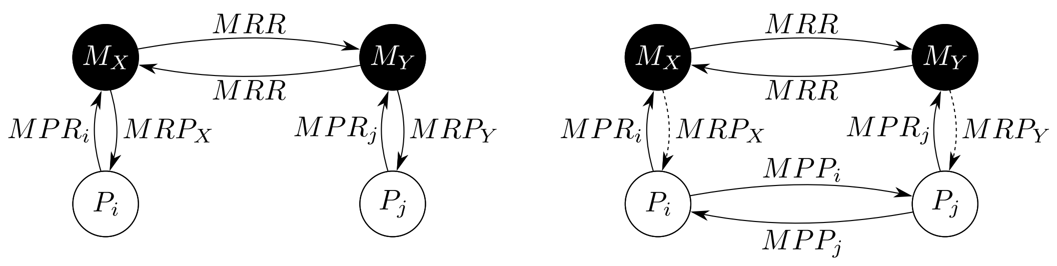


Figure 4.1: MRACS architecture.

Formally, MRACS comprises three entities: a set of m mirrors $M = \{M_f \mid M_f$ is a mirror with a unique ID $f\}$, where each mirror M_f runs a referee; a set of n players $P = \{P_i \mid P_i$ is a player with a unique ID $i\}$; and an authentication server S_A.

As in MS [41], MRACS requires mirrors to be connected via a private, low-delay, low-loss rate, and multicast enabled network. Due to the cost overhead of provisioning a mirror [85, 100] (security, cooling, power, *etc.*), we assume the publisher provisions a small number of mirrors, each comprised of a cluster of hosts capable of simulating the entire virtual world. Similar to RACS, each player sends updates to his Ingress-mirror (I-mirror) and every player with whom he is interacting in PP mode (connected lines). Updates are sent through mirrors using PRP mode if players are not in each others' AoI, direct communication is not possible, or cheating is suspected. To maintain consistency among the mirrors, every mirror forwards all updates received from clients to all other mirrors, Egress-mirrors (E-mirrors), through the private network, and all mirrors simulate the update. Like MS, MRACS assumes well-provisioned mirrors with a low probability of failure. If a mirror crashes, then the remaining mirrors will continue the game, and players disconnected from the failed mirror must re-connect to a functioning mirror.

Server S_A authenticates joining players, assigns each player a unique ID and an I-mirror (client-to-mirror assignment - see Section 4.2), and adjusts the round length τ. Server S_A divides game time into rounds of length τ within which every player generates an update and sends it to his I-Mirror and players (described in Section 4.1.2). A late message (not received within its round) is considered for a future round if no newer messages have been received; otherwise it is discarded. Game time is synchronised among all mirrors and clients using NTP [97], and rounds may be pipelined [64] to improve responsiveness. To adjust the round length τ, mirrors periodically request information from players about the peer-to-peer delays and forward it to S_A, which calculates the new round length as in Section 3.3. As interacting players may be connected to different mirrors the round length must be set globally for all mirrors and players.



$$\begin{tikzcd}
M_X \arrow[r, bend left, "MRR"] \arrow[d, bend left, "MRP_X"] & M_Y \arrow[l, bend left, "MRR"] \arrow[d, bend left, "MRP_Y"] \\
P_i \arrow[u, bend left, "MPR_i"] & P_j \arrow[u, bend left, "MPR_j"]
\end{tikzcd}$$

(a) PRP Mode.

$$\begin{tikzcd}
M_X \arrow[r, bend left, "MRR"] \arrow[d, bend left, dotted, "MRP_X"] & M_Y \arrow[l, bend left, "MRR"] \arrow[d, bend left, dotted, "MRP_Y"] \\
P_i \arrow[u, bend left, "MPR_i"] \arrow[r, bend left, "MPP_i"] & P_j \arrow[u, bend left, "MPR_j"] \arrow[l, bend left, "MPP_j"]
\end{tikzcd}$$

(b) PP Mode.

Figure 4.2: MRACS communication modes.

4.1.2 MRACS Communication Modes

MRACS communication modes are similar to those of RACS, described in Section 3.2.2. As shown in Figure 4.2, the communication between each pair of mutually aware players, P_i assigned to M_X and P_j assigned to M_Y, can be direct (Peer-Peer: PP mode) or routed through their I-Mirrors (Peer-Referee-Peer: PRP mode). Note, P_i and P_j may have the same or different I-Mirrors (*i.e.,* $X = Y$ or $X \neq Y$).

MRACS considers four different message formats:

1. peer to peer message - $MPP_i(U_i)$, signed by the sender P_i,
2. peer to referee message - $MPR_i(U_i, S_i, V_i, H_i, T_i)$, signed by the sender P_i,
3. referee to peer message - $MRP_f(i, U_i', C_i)$, signed by the sender M_f, and
4. referee to referee message - $MRR(i, U_i', S_i)$.

The MPP, MPR, and MRP messages are identical to those in RACS; see Section 3.2.1 for the details. Note, after an MPR message is received and validated - correct digital signature and a fresh round number - U_i is timestamped with the I-mirror's current round number, r', to produce U_i'. This is used for event ordering between mirrors to maintain consistency, as the player's round number cannot be trusted. As in MS, we assume the private network is secure, loss-less, and multicast enabled; hence, MRR is not signed. Mirrors exchange MRR messages to maintain consistent state and validate the simulation. For each received MPR the receiving mirror multicasts an MRR message containing the player's identity, update, and secret information to all other mirrors. The I-mirror is responsible for validating the MPP's digital signature and ensuring that the round number is greater than all previously received MPP messages. If either check fails the MPP is discarded. These checks are performed by the I-mirror so that they are only performed once (reducing processing overhead) and to reduce the size of MRR messages (reducing the Mirror-to-Mirror bandwidth requirements).

The PP and PRP modes in MRACS are similar to those in RACS as they utilise the same QoS requirements between players. However, as there are many small changes to support multiple referees, we include a complete description of the MRACS protocol for clarity. When two mutually aware PRP players, P_i assigned to M_X and P_j assigned to M_Y ($X = Y$ or $X \neq Y$), are within each others' AoI, M_X sends an MRP containing C_j to P_i, and M_Y sends an MRP containing C_i to P_j, transitioning P_i and P_j to PP mode. On the other hand, P_i reverts to PRP (with respect to P_j) if: (i) P_j is no longer in P_i's AoI or vice-versa; (ii) P_i receives less than p percent of P_j's last $s \geq 1$ messages, or (iii) P_i does not receive P_j's update for more than $w \geq 0$ consecutive rounds. Requirement (i) provides AoI filtering to reduce bandwidth; requirement (ii) ensures that a minimum percentage of updates are received, preventing a cheater repeatedly sending one message and then dropping $w-1$ consecutive messages; while requirement (iii) ensures that losses are not clustered, which would have a large impact on the game-play experience. For either case (ii) or (iii), P_i sends an MPP_i to P_j and an MPR_i to M_X notifying them of the reversion. Mirror M_X forwards this to M_Y, which only forwards P_i's moves to P_j if P_i is within P_j's AoI. As in RACS (Chapter 3), MRACS is cheat-proof when $w = 0$ or $p = 100\%$. For optimal game play the values for w, p, and s should balance cheat prevention with minimising PP to PRP reversions.

The steps performed by each peer in every round are identical to those in RACS, and are summarised in Algorithm 3.1. Algorithm 4.1 shows the corresponding steps for each mirror; they are similar to the referee's steps in Algorithm 3.2, except for lines 2 and 25 to 29.

Note, to prevent the time-stamp cheat, the round number in MPR and MPP messages must not be used for event ordering. This may result in players perceiving game anomalies, caused by packets arriving late or out of order; however, this approach maximises responsiveness while preventing cheating [5, 43].

4.1.3 MRACS Synchronisation

As in MS, MRACS requires a synchronisation mechanism (*e.g.*, TSS or BS - see Section 2.3.2.2) between mirrors. However, unlike in MS, we expect most players will use PP mode in which they exchange updates directly, and thus the synchronisation delay in the mirrors is less critical in MRACS than in MS. Due to its lower processing and memory overhead, we recommend BS for synchronising mirrors in MRACS. Further, unlike TSS, BS avoids rollbacks, which are process intensive and cause temporal anomalies. The additional Δ delay in mirrors will only influence PRP players; hence, providing additional motivation for players to use PP mode.

4.1.4 MRACS Security

Table 4.1 summarises the strategies that MRACS uses to solve each cheat. As MRACS includes all security measures used by RACS, it prevents or detects all known protocol-level cheats, information exposure, and the invalid command cheat. However, the steps required to prevent the inconsistency (checking the hashes of MPP messages), spoofing

Algorithm 4.1: MRACS_referee_game_loop()

```
/* The game loop run by each mirrored referee, $M_f$, in every round. */;
begin
    for every player $P_i$ whose I-mirror is $M_f$ do
        if $P_i$ attempted an invalid action (due to inconsistent state) then
            Send an $MRP_f$ to $P_i$ with the current game state.
        end
        for every $P_j$ in PRP communication with $P_i$ do
            if $P_i$ and $P_j$ should enter PP mode. then
                Send $MRP_f(j, U'_j, C_j)$ to $P_i$.
            else
                Send $MRP_f(j, U'_j)$ to $P_i$.
            end
        end
    end
    for every $MPR_i$ received within the round do
        Use $C_i$ to validate $MPR_i$'s digital signature.
        Discard $MPR_i$ if validation fails.
        Compute $H'_i$ by XORing all $H(U_j)$ for every $MPP_j$ $P_i$ received on-time in the
        previous round (indicated by $V_i$).
        Compute $H(U_j)'$ for every $MPP_j$ $P_i$ received late in the previous round
        (indicated by $T_i$).
        if $H'_i \neq H_i$ then
            Request $P_i$ to forward all $MPP_j$ messages used to calculate $H_i$.
        end
        for every $H(U_j)' \neq H(U_j)$ do
            Request $P_i$ to forward $MPP_j$.
        end
        for every $MPP_j$ forwarded by $P_i$ do
            Use $C_j$ to validate $MPP_j$'s digital signature.
            if $MPP_j$'s signature is invalid then
                $P_i$ is attempting to frame $P_j$.
            else if $MPP_j$'s $U_j$ does not match that in $MPR_j$ then
                $P_j$ is attempting the inconsistency cheat.
            end
        end
        Use the $r$ in $MPR_i$ to discard old updates from each player.
        Timestamp the $U_i$ in $MPR_i$ with $r'$ to create $U'_i$.
        Multicast $MRR(i, U'_i, S_i)$ to all E-mirrors.
    end
    for every MRR received within the round do
        Use the $r'$ in each $U'_i$ to discard old updates from each player.
    end
    Simulate all $U'_i$ using $r'$ for event ordering.
    Use interpolation/extrapolation for missing updates.
end
```

Cheats Prevented	**MRACS Strategy**
Information Exposure	On-demand Loading
Invalid Command	Referee simulation and validation
Suppressed Update	Referee authoritative state
Replay	Round number r validated by the I-mirror
Spoofing	Signed messages validated by the I-mirror
Undo	No commit / reveal steps
Timestamp	Updates timestamped by the I-mirror
Cheats Detected	**MRACS Strategy**
Bugs	Software patches
RMT / Power Levelling	Statistical analysis of log files
Bots / Reflex Enhancers	Combine MRACS with a CDS
Fixed Delay	QoS requirements and reversion to PRP
Inconsistency	I-Mirrors validate message signatures and hashes
Blind Opponent	Reversion to PRP mode
Undetectable Cheats	**MRACS Strategy**
Collusion	None
Proxies / Reflex Enhancers	None

Table 4.1: Summary of MRACS cheat detection/prevention strategies.

(validating the digital signature of received MPP messages), and replay (discard messages with old round numbers) cheats are performed by the I-mirrors, so that each MPR message is validated once only.

Unlike RACS, mirrors in MRACS must use round numbers for message ordering. To prevent the timestamp cheat each I-mirror timestamps received MPR messages with its current round number, r', as shown in Algorithm 4.1, line 25. All other cheats are detected / prevented as in RACS (Section 3.4.1).

4.2 Client to Mirror Assignment (CMA)

The CMA problem is to assign clients to mirrors such that the client-to-mirror delay is minimised, and no mirror is overloaded. It assumes each mirror has fixed bandwidth and processing resources. To accommodate player joins and leaves, we introduce a pair of optimal algorithms and a pair of faster, but sub-optimal, algorithms.

Let L_S denote the processing required to simulate the virtual world for each player, and L_C denote the processing required to send and receive updates and to perform AoI filtering for each player. Note that communicating updates and AoI filtering are a significant portion of a server's workload [5]. We assume each player requires at most A bandwidth and $L_s + L_c$ processing resources of the mirror. Every mirror M_j provides fixed bandwidth α_j and processing power β_j. Therefore, M_j has capacity to support a fixed number of players C_j. For bandwidth, since every player requires up to A units, M_j can support $\frac{\alpha_j}{A}$ clients. From the processing perspective, every mirror requires at most $n \times L_s$ units of processing power to simulate the virtual world involving all n players in P; thus, M_j can support $\frac{\beta_j - n \times L_s}{L_c}$ players. Therefore, the capacity of M_j is calculated as $C_j = min\{\frac{\alpha_j}{A}, \frac{\beta_j - n \times L_s}{L_c}\}$.

If players typically use less than A bandwidth and $L_s + L_c$ processing resources, the mirrors' resources will be underutilised. To improve resource utilisation, the publisher may set $C_j > min\{\frac{\alpha_j}{A}, \frac{\beta_j - n \times L_s}{L_c}\}$; however, if the processing or bandwidth resources of the mirror are exceeded the users' experience rapidly deteriorates. As each mirror supports hundreds of players [85], we expect variations between players' resource usages will average out. Further, it is now considered standard practise to have a long beta test phase (six months or more) to accurately measure the client workload and modify the system for maximum scalability [100].

We define the delay $d_{i,j}$ as the difference between the time a mirror M_j receives an update and the time it was sent by client P_i; we assume the delay from M_j to P_i is also $d_{i,j}$. The delay experienced by a player has considerable impact on his enjoyment [32, 56]; therefore, it is desirable to minimise delay. Note that the delay between mirrors is not considered in our model as it is not affected by the player assignment.

4.2.1 CMA Problem Statement

Let DM denote a delay matrix of size $n \times m$ for all n clients in P and m mirrors in M. An element in row i and column j of DM, $DM[i, j]$, represents delay $d_{i,j}$ for P_i and M_j, for all $i = 1, 2, ..., n$ and $j = 1, 2, ..., m$. Let CMA_n be a set of all client-to-mirror assignments for DM. We define CMA_n optimal if $D = \sum_{P_i \in P, M_j \in M} d_{i,j} \times x_{i,j}$ is minimised, subject to constraints:

(i) $\forall P_i \in P, \sum_{M_j \in M} x_{i,j} = 1$, and

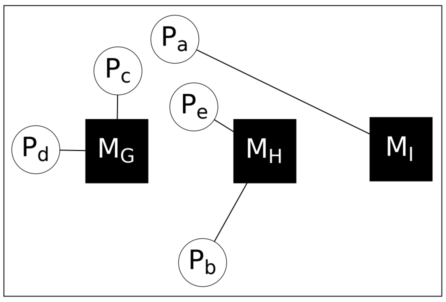
(ii) $\forall M_j \in M, \sum_{P_i \in P} x_{i,j} \leq C_j$;

where $x_{i,j} \in \{0, 1\}$ is equal to 1 if P_i is assigned to M_j, and 0 otherwise. Constraint (i) ensures that every player is assigned to a mirror, and constraint (ii) avoids saturating a mirror. We assume fixed M and dynamic P (due to player joins and leaves). The CMA-J problem is to construct an optimal CMA_{n+1} from an optimal CMA_n, including the assignment of a joining P_i. Similarly, the CMA-L problem is to construct an optimal CMA_{n-1} from an optimal CMA_n, including the removal of a leaving P_i. Note that as the workload generated by each player is not necessarily equal, an optimal assignment does not guarantee that a mirror is not saturated or underutilised. As discussed in Section 4.2, however, we anticipate that resource usage will be close to optimal.

Both CMA-J and CMA-L may require reassigning players to different mirrors to minimise delay. As every mirror stores and simulates the entire virtual world, MRACS is well suited to transfer players between mirrors, as no game state must be passed. Further, as most games use the User Datagram Protocol (UDP) for communication, there is little or

	M_G	M_H	M_I
P_a	4	5	$\underline{10}$
P_b	4	$\underline{5}$	10
P_c	$\underline{1}$	8	10
P_d	$\underline{1}$	8	10
P_e	2	$\underline{4}$	10

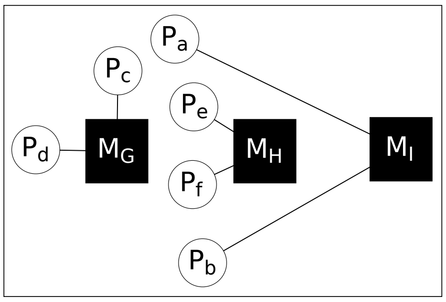
(a) Delay matrix *DM*.



(b) Optimal CMA$_5$.

	M_G	M_H	M_I
P_a	4	5	$\underline{10}$
P_b	4	5	$\underline{10}$
P_c	$\underline{1}$	8	10
P_d	$\underline{1}$	8	10
P_e	2	$\underline{4}$	10
P_f	2	$\underline{4}$	10

(c) *DM* with joining P_f.



(d) Optimal CMA$_6$.

Figure 4.3: Example client and mirror configurations for the CMA problem.

no overhead to establish a new connection. We believe the benefits of transferring players to reduce the average game delay far outweigh the costs, as the bandwidth and processing required to perform a hand off are far below that required to process, simulate, and send updates about the game. The responsibility of reassigning players to mirrors belongs to the authentication server S_A as it stores *DM*.

In delay sensitive games such as FPS, players compensate for high delays by modifying their playing style. However, this is difficult if their delay fluctuates wildly due to mirror reassignments. As the CMA problem is to minimise *D*, it is unlikely that the problem's solution would reassign a player to a mirror with significantly higher delay. Our simulation results in Section 4.3.3.2 support this.

Figure 4.3 shows an example of the CMA problem for three mirrors: M_G, M_H, and M_I, with capacity $C_G = C_H = C_I = 2$, and five clients: P_a, P_b, P_c, P_d, and P_e. Each number in Figure 4.3(a) represents the delay from a client (rows) to a mirror (columns). Figure 4.3(b) shows the optimal CMA_5 for *DM* in Figure 4.3(a); the cost of each client assignment is underlined in Figure 4.3(a). The total, average, and maximum delays of the solution are 21, 4.2, and 10, respectively. For the joining player P_f with $d_{f,G} = 2$, $d_{f,H} = 4$, and $d_{f,I} = 10$, the CMA-J problem is to obtain CMA_6 from CMA_5 that minimises delay *D* subject to constraints (i) and (ii). Figure 4.3(c) shows the updated *DM* and Figure 4.3(d) gives the optimal CMA_6. The optimal result requires reassigning client P_b from M_H to M_I and assigning P_f to M_H. Similarly, considering the optimal CMA_6 in Figure 4.3(d), and leaving player P_f, the solution to the CMA-L problem results in the assignment in Figure 4.3(b).

4.2.2 CMA Algorithms

When a player joins, the authentication server S_A verifies the player for subscription, banning, *etc*., and then transmits the list of mirrors to the client. The client uses echo messages to measure the delay to each mirror and sends the results to S_A. The server uses a CMA-J algorithm to determine a mirror for the client; it notifies the client and may ask other clients to transfer to different mirrors. As players leave, the average client-to-mirror delay of the remaining players may no longer be optimal. For each leaving player, S_A uses a CMA-L algorithm to determine the optimal client-to-mirror assignments for the remaining players, and may instruct multiple remaining clients to transfer to different mirrors.

The following subsections describe two CMA-J algorithms (J-SA and J-Greedy) and two CMA-L algorithms (L-SA and L-Greedy).

4.2.2.1 CMA-J Algorithms

Our J-SA considers CMA-J as a special case of the Terminal Assignment (TA) problem [81], described in Section 2.3.2.4. The TA problem is to determine the assignment of each terminal to a concentrator such that no concentrator exceeds its capacity and the overall system delay is minimised; this problem is NP-complete [82, 128]. Note that the TA problem assumes fixed numbers of concentrators and terminals and they are known in advance, which is different to ours as players join and leave the game without warning. In our model, every client (terminal in TA) has equal weight since we assume that each player consumes the same amount of bandwidth and processing power from its I-mirror (concentrator in TA). This special case can be optimally solved in polynomial time using either the Sequential Assignment (SA) [129] or the Alternating Chain [81] algorithms. We use SA, described in Section 2.3.2.4, for our J-SA.

Algorithm 4.2 shows our J-SA, implemented using a transfer heaps array TH[1...m, 1..m] which is updated for each player join. Note that J-SA also uses Algorithms 4.3-4.6. Each element TH[X, Y] is a min heap that contains tuples (*cost*, *player*); cost is the delay offset $(d_{i,Y} - d_{i,X})$ of transferring P_i from M_X to M_Y. The heap in TH[X, Y] contains at most C_X tuples; initially each TH[X, Y] contains a *null* heap. There are two cases for each joining player P_i. In case one, the closest mirror M_X has spare capacity; thus, the algorithm simply assigns P_i to M_X and updates the heaps in TH for all possible reassignments of P_i. In case two, M_X is full, and the optimal assignment may require a sequence of player reassignments (called a chain). For this case, the algorithm comprises two main steps, Step 1 and Step 2.

Step 1 computes the cost of assigning P_i to each of the full-capacity mirror M_F, and the cost of reassigning a player from each M_F to another mirror M_Y. Since M_Y may also be full, this step also computes the cost of reassigning a player from M_Y to another mirror, *etc*. In essence, this step creates all possible chains of reassignments to make space for P_i; each chain terminates when a player is reassigned to a mirror with spare capacity. Function construct_labels_join() in Step 1 builds the chains, and stores them in array Label[1...m].

Algorithm 4.2: J-SA(P_i, $DM[1...n, 1...m]$)

```
/* $P_i$ - The joining player */
/* $DM[1...n, 1...m]$ - The delay matrix */
begin
    $X$ = index_min($DM[i]$) /* index of the mirror with minimum cost */
    if $M_X$ is not full then
        assign($P_i$, $M_X$, $DM$)
    else
        /* Step 1 */
        Label = construct_labels_join(TH) /* Algorithm 4.6 */
        for every $M_Y \in M$ do
            add $DM[i,Y]$ to Label[$Y$].cost
        end
        $Z$ = index_min(Label) /* index of the label with minimum cost */
        /* Step 2 */
        if $\exists$ $M_Y$ with spare capacity and $DM[i,Y]$ = Label[$Z$].cost then
            assign($P_i$, $M_Y$, $DM$) /* Algorithm 4.4 */
        else
            assign($P_i$, $M_Z$, $DM$) /* Algorithm 4.4 */
            transfer_players(Label, TH, $Z$) /* Algorithm 4.3 */
        end
    end
end
```

Algorithm 4.3: transfer_players(Label[*1...m*], TH[*1...m,1...m*], *from*)

```
/* Label[$1...m$] - Array of labels*/
/* TH[$1...m, 1...m$] - Transfer Heaps array*/
/* $from$ - Mirror at the start of the chain */
begin
    repeat
        $to$ = Label[$from$].$mirror$
        $j$ = TH[$from$, $to$].$player$
        remove($P_j$) /* Algorithm 4.5 */
        assign($P_j$, $M_{to}$, $DM$) /* Algorithm 4.4 */
        $from = to$
    until $M_{from}$ does not exceed its capacity
end
```

Algorithm 4.4: assign(P_i, M_X, $DM[1...n, 1...m]$)

```
/* $P_i$ - Player to assign */
/* $M_X$ - I-Mirror to assign $P_i$ too. */
/* $DM[1...n, 1...m]$ - The delay matrix */
begin
    set $M_X$ as $P_i$'s mirror
    reduce $M_X$'s available capacity by 1
    for every $M_Y \in M$ where $Y \neq X$ do
        insert(TH[$X,Y$], DM[$i,Y$] - DM[$i,X$], $P_i$)
    end
end
```

Algorithm 4.5: remove(P_i, M_X)

```
/* $P_i$ - Player to assign */
/* $M_X$ - $P_i$'s assigned I-Mirror */
begin
    increase $M_X$'s available capacity by 1
    for every $M_Y \in M$ where $Y \neq X$ do
        delete(TH[$X,Y$], $P_i$'s tuple)
    end
end
```

Algorithm 4.6: construct_labels_join(TH[*1...m,1...m*])

```
/* TH[$1...m, 1...m$] - Transfer Heaps array */
/* Exports Label[$1...m$] - The array of labels */
begin
    for every $M_Y \in M$ where $M_Y$ is full do
        Label[$Y$].cost = $\infty$
    end
    for every $M_Y \in M$ where $M_Y$ is not full do
        Label[$Y$].cost = 0
    end
    while labels were changed do
        for every $M_F \in M$ where $M_F$ is full do
            for every $M_Y \in M$ where $Y \neq F$ do
                if Label[$Y$].cost + TH[$F,Y$].cost < Label[$F$].cost then
                    Label[$F$].cost = Label[$Y$].cost + TH[$F,Y$].cost
                    Label[$F$].mirror = $Y$
                end
            end
        end
    end
end
```

	G	***H***	***I***
G	-	(7, c)	(9, c)
H	(-2, e)	-	(5, b)
I	(-6, a)	(-5, a)	-

(a) TH for Figure 4.3(a).

Label[G]	=	(∞, -)
Label[H]	=	(∞, -)
Label[I]	=	(0, -)

(b) Initial Label.

Label[G]	=	(9, I)
Label[H]	=	(5, I)
Label[I]	=	(0, -)

(c) Final Label.

	G	***H***	***I***
G	-	(7, c)	(9, c)
H	(-2, e)	-	(6, e)
I	(-6, a)	(-5, a)	-

(d) TH for Figure 4.4(c).

Figure 4.4: Example of CMA-J using J-SA.

Each Label[Y] is a pair (*cost*, *mirror*) representing a chain starting from M_Y; cost is the minimum additional delay (called chain cost) of assigning P_i to M_Y, and *mirror* is the next mirror in the chain after M_Y. Note that a chain can be a sub-chain of others. Each Label[Y] is initialised to (∞, -) if M_Y is full and to (0, -) if it has spare capacity. Function construct_labels_join() builds the chains as follows. For each full mirror $M_F \in M$, Label[F] is updated to the minimum cost of transferring a player from M_F to each $M_Y \in M$ $(Y \neq F)$, including the chain cost at M_Y (*i.e.*, $\text{Label}[F].cost = min\{\text{Label}[Y].cost + \text{TH}[X,Y].cost\}$). The algorithm repeatedly updates the chain for each label until no labels are updated. No cycles occur because a chain must end at a non-full mirror, since full mirrors start with an initial cost of ∞. After executing construct_labels_join(), Step 1 of J-SA continues by adding to each Label[Y].$cost$ the delay of assigning P_i to M_Y $(d_{i,Y})$, completing the delay of each chain. The minimum delay chain is the Label[Z] with the smallest cost.

In Step 2 of J-SA, the algorithm assigns P_i to M_Y if there exists a non-full mirror M_Y with $d_{i,Y}$ equal to the delay of the minimum chain, Label[Z].$cost$. Otherwise, player reassignments are performed by tracing the minimum delay chain from M_Z using the transfer_players() function. In either case, TH is updated. The array Label is used to trace the mirrors in the chain, and array TH is used to find the corresponding players to reassign.

The time complexity of J-SA is calculated as follows. Function construct_labels_ join() takes $O(m^3)$ time. The insert() and delete() heap operations use $O(log(x))$ time, where x is the size of the heap. As $x \leq n$, assign() and remove() take $O(m \times log(n))$ time; hence, transfer_players() runs in $O(m^2 \times log(n))$ time since the longest possible chain is of length m. Therefore, the time complexity of J-SA is $O(m^2 \times log(n) + m^3)$.

We demonstrate J-SA using the example in Figure 4.3 with joining P_f. Since M_G (the closest mirror to P_f) is full, a chain of reassignments may be required. Figure 4.4(a) shows the root tuple of each heap in array TH for the optimal CMA in Figure 4.3(b), while Figure 4.4(b) shows the initialised contents of Label. Since M_I is not full, the algorithm considers moving players only from M_G and M_H. In the first iteration, there is only one possible chain from M_G, *i.e.*, transferring P_c to M_I at a cost of $\text{Label}[I].cost + \text{TH}[G,I].cost = 0 + 9 = 9$

(Label$[G]$.*cost* $= 9$ and Label$[G]$.*mirror* $= I$). For M_H, transferring P_b to M_I (with cost $0+5=5$) is better than P_e to M_G (with cost $9-2=7$), and thus the chain to M_I is selected (Label$[H]$.*cost* $=5$ and Label$[H]$.*mirror* $= I$). In the second iteration, the chain from M_G to M_H (*i.e.*, P_c to M_H) with cost $5+7=12$ is considered but discarded as it is greater than the existing chain to M_I with cost 9. Similarly, for M_H, the existing chain is optimal, and thus the algorithm exits the while loop as no labels were updated. From the computed labels, and DM in Figure 4.3(c), the algorithm computes the cost of the chains at M_G, M_H, and M_I as: Label$[G]$.*cost* $+ d_{f,G} = 9+2=11$, $5+4=9$, and $0+10=10$, respectively. The algorithm finds the min cost of 9 for M_H. Since there is no mirror with spare capacity and delay 9 from P_f, the algorithm follows the minimum cost chain of player reassignments. First, P_f is assigned to M_H, and Label$[H]$.*mirror* $= I$ gives the next mirror in the chain. Tuple TH$[H,I]$.*player* gives P_b, the player to reassign from M_H to M_I. As M_I has spare capacity the chain ends. Figure 4.3(d) shows the resulting optimal assignment, and Figure 4.4(d) shows the corresponding TH.

Algorithm 4.7 is J-Greedy, which is faster than J-SA and is able to produce nearly optimal client assignments. Note that J-Greedy uses Algorithm 4.4 to assign players to mirrors. The greedy algorithm is an extension to Cronin *et al.* [43] that considers mirror capacity. J-Greedy sorts all mirrors in increasing delay from the joining player and selects the closest mirror with spare capacity. Conceptually, J-Greedy is J-SA but considering only chains of length 0. If all mirrors have spare capacity, then J-Greedy will produce optimal results; however, as observed by Kershenbaum [81], if the system is close to full capacity, then this algorithm produces poor results, as later joining players are assigned to mirrors with high delays. The complexity of J-Greedy is as follows. Sorting the mirrors requires $O(m \times log(m))$ time. Note that L-Greedy (Section 4.2.2.2) that is used with J-Greedy requires the TH array. Therefore, J-Greedy must maintain array TH. As in J-SA, updating TH requires $O(m \times log(n))$ time; thus, the time complexity of J-Greedy is $O(m \times log(n))$.

Algorithm 4.7: J-Greedy(P_i, $DM[1...n, 1...m]$)

```
/* $P_i$ - The leaving player */
/* $DM[1...n, 1...m]$ - The delay matrix */
begin
    sort $DM[i]$ by delay in ascending order
    for each $Y \in DM[i]$ do
        if $M_Y$ has spare capacity then
            assign($P_i$, $M_Y$, $DM$) /* Algorithm 4.4 */
            break
        end
    end
end
```

4.2.2.2 CMA-L Algorithms

To obtain optimal delays for CMA-L, one may use a brute-force approach that runs J-SA on each of the remaining players. In contrast, the solution in Cronin *et al.* [43] (we call it L-Ignore) keeps the client-to-mirror assignment of the remaining clients, and hence is not optimal. The SA algorithm in reference [129] cannot be directly used to solve CMA-L. Therefore, we propose two novel algorithms, the optimal L-SA and the faster heuristic L-Greedy.

L-SA in Algorithm 4.8 is similar to J-SA, and uses Algorithms 4.3-4.5 and 4.9. When a player P_i leaves, L-SA maintains array TH and increments the available capacity of P_i's mirror M_X by 1. If the resulting capacity is greater than 1 (*i.e.*, M_X was not full), no player reassignments are required since the CMA remains optimal. Else, L-SA performs two steps, which are similar to those in J-SA. The construct_labels_leave() function in L-SA builds the chains starting from each mirror M_O ($O \neq X$ and M_O is occupied; *i.e.*, not empty) and ending at M_X; in contrast, J-SA builds chains starting at a full mirror and ending at a non-full mirror. In L-SA, the function does not transfer players to non-full mirrors (except for M_X) as this would not decrease the system delay. As in J-SA, Step 2 of L-SA uses transfer_players() to perform the chain of reassignments; however, the chain is only performed if the total system delay will be reduced; *i.e.*, the minimum chain cost is below zero. Similar to those for J-SA, construct_labels_leave() and transfer_players() have time complexities of $O(m^3)$ and $O(m^2 \times log(n))$, respectively. Thus, the time complexity of L-SA is $O(m^2 \times log(n) + m^3)$, equal to that of J-SA. The proof that L-SA is optimal is included in Appendix A.

Algorithm 4.8: L-SA(P_i, M_X, $DM[1...n, 1...m]$)

```
/* $P_i$ - The leaving player */
/* $M_X$ - $P_i$'s assigned I-Mirror */
/* $DM[1...n, 1...m]$ - The delay matrix */
begin
    remove($P_i$)
    if $M_X$'s available capacity is 1 then
        /* Step 1 */
        Label = construct_labels_leave(TH, $M_X$) /* Algorithm 4.9 */
        Z = index_min(Label) /* index of the label with minimum cost */
        /* Step 2 */
        if Label[Z].cost < 0 then
            transfer_players(Label, TH, Z) /* Algorithm 4.3 */
        end
    end
end
```

We demonstrate L-SA with an example. Assume that player P_c from Figure 4.3(d) leaves; the resulting *DM* is shown in Figure 4.5(a); the player assignments are under-

Algorithm 4.9: construct_labels_leave(TH[*1...m,1...m*], *l*)

```
/* TH[$1...m, 1...m$] - Transfer Heaps array */
/* $X$ - $M_X$ is the leaving player's I-Mirror */
/* Exports Label[$1...m$] - The array of labels */
begin
    for every $M_Y \in M$ where $Y \neq X$ do
        Label[$Y$].cost = $\infty$
    end
    Label[$X$].cost = 0
    while labels were changed do
        for every $M_O \in M$ where $M_O$ is not empty and $O \neq X$ do
            for every $M_Y \in M$ where $Y \neq O$ and ($Y = X$ or $M_Y$ is full) do
                if Label[$Y$].cost + TH[$O,Y$].cost < Label[$O$].cost then
                    Label[$O$].cost = Label[$Y$].cost + TH[$O,Y$].cost
                    Label[$O$].mirror = $Y$
                end
            end
        end
    end
end
```

lined. Figure 4.5(b) shows the TH corresponding to 4.5(a). Initially Label$[G]$.*cost* $=$ 0, Label$[H]$.*cost* $= \infty$,and Label$[I]$.*cost* $= \infty$; the label's mirrors are uninitialised. For M_H, the chain to M_G with cost Label$[G]$.*cost* $+$ TH$[H,G]$.*cost* $= 0 - 2 = -2$ is selected (Label$[H]$.*cost* $= -2$, Label$[H]$.*mirror* $= G$). For M_I, player P_a can be moved to M_H with a delay reduction of Label$[H]$.*cost* $+$ TH$[I,H]$.*cost* $= -2 - 5 = -7$ (Label$[I]$.*cost* $= -7$, Label$[I]$.*mirror* $= H$). The second iteration does not change any labels and thus the algorithm exits the while loop. As Label$[I]$ has the minimum cost, the chain begins at I, and the sequence of reassignments is: P_a to M_H, P_e to M_G.

	M_G	M_H	M_I
P_a	4	5	$\underline{10}$
P_b	4	5	$\underline{10}$
P_d	$\underline{1}$	8	10
P_e	2	$\underline{4}$	10
P_f	2	$\underline{4}$	10

(a) *DM* after P_c leaves.

	G	***H***	***I***
G	-	(7, d)	(9, d)
H	(-2, *e*)	-	(6, *e*)
I	(-6, *a*)	(-5, *a*)	-

(b) TH for Figure 4.5(a).

Label$[G]$	=	(0, -)
Label$[H]$	=	(∞, -)
Label$[I]$	=	(∞, -)

(c) Initial Label.

Label$[G]$	=	(0, -)
Label$[H]$	=	(-2, G)
Label$[I]$	=	(-7, H)

(d) Final Label.

Figure 4.5: Example of CMA-L using L-SA.

Algorithm 4.10, L-Greedy, is faster than L-SA and produces nearly optimal results. Note that L-Greedy uses Algorithms 4.4 and 4.5. The L-Greedy heuristic reassigns the player that will give the maximum benefit. As shown in Algorithm 4.10, assuming a leaving player P_i assigned to M_X, L-Greedy selects the client with the largest delay reduction when reassigned to M_X. This is repeated until a client is reassigned from a non-full mirror or there is no client that will benefit from being reassigned. Conceptually, L-Greedy is similar to L-SA, but builds only a single chain of reassignments by greedily selecting the most beneficial reassignment at each step. The worst case run time is $O(m^2 \times log(n))$, equal to L-SA without Step 1. Note that the CMA solution that dynamically allows players joining and leaving must comprise both a CMA-J and CMA-L. Notice that J-SA requires optimal CMA_n to produce CMA_{n+1}, and that L-SA requires optimal CMA_n to produce CMA_{n-1}. Therefore, J-SA and L-SA cannot be combined with a non-optimal CMA-L or CMA-J algorithm, respectively.

Algorithm 4.10: L-Greedy(P_i, M_X, $DM[1...n, 1...m]$)

```
/* $P_i$ - The leaving player */
/* $M_X$ - $P_i$'s assigned I-mirror */
/* $DM[1...n, 1...m]$ - The delay matrix */
begin
    remove($P_i$) /* Algorithm 4.5 */
    if $M_X$ was full then
        continue = true
        repeat
            $min = \infty$
            for every $M_Y \in M$ where $Y \neq X$ and $M_Y$ is not empty do
                if TH[Y,X].cost < min then
                    min = TH[$Y$,$X$].cost
                    $F = Y$ /* $M_F$ is the mirror to move from */
                end
            end
            if $min \geq 0$ then
                continue = false
            else
                if $M_f$ is not full then
                    continue = false;
                end
                $P_j$ = TH[$F$,$X$].player
                remove($P_j$) /* Algorithm 4.5 */
                assign($P_j$, $M_X$, $DM$) /* Algorithm 4.4 */
                $X = F$
            end
        until continue = false
    end
end
```

4.3 Performance Analysis

4.3.1 Bandwidth and Processing - Analytical Evaluation

We compare the performances of RACS, MS, and MRACS in terms of their clients' and mirrors' in-bandwidth and out-bandwidth requirements for n clients and m mirrors. We use the notation from Section 3.4.2; *i.e.*, a avatars within each player's AoI, o overhead per message, and the |MPR|, |MRP|, and |MPP| message sizes. Several studies [31, 55, 83, 127] have shown that the mean and standard deviation of player updates is small; thus, the MRR message size, |MRR|, has small mean and standard deviation. Like in MRP, we assume each MRR message aggregates multiple player updates to reduce the bandwidth consumed by packet overheads.

Table 4.2 shows the results, where public (private) denotes communication through the Internet (private network). Note, if all players in MRACS use PRP mode (worst case), then MRACS bandwidth equals MS.

When all pairs of players use PP mode (best case), the mirrors' outgoing bandwidth is negligible, as they must only notify players to transition to PP mode. However, players' outgoing bandwidth increases linearly with the number of PP players. Due to the success of P2P networks, and as game clients have low bandwidth requirements [55], we believe this increase will not exceed most players' capacity. Further, lower delay will motivate players to use PP. Note that a is directly related to n (*i.e.*, an increase in n causes an increase in a); thus, the outgoing bandwidth in MS grows faster than linearly for n, potentially causing a bottleneck. MRACS in PP mode solves this bottleneck as its bandwidth grows only linearly for n. Cronin *et al.* [41] showed MS superior to C/S with respect to server/mirror bandwidth requirements, and thus we may conclude that MRACS is also superior to C/S in the same respect. Finally, as MRACS distributes the bandwidth across m mirrors it is more scalable than RACS.

We analytically compared the processing requirements of TSS and BS. Note that every update in TSS with l states is processed l times, and more when rollbacks occur. Consider an update processing time of u, with a β probability of causing a rollback requiring v processing time. Thus, the cost of TSS is: $(l \times u + \beta \times v) \times \lambda \times n$, where λ is the number of updates generated by each player per second. On the other hand, BS processing cost is only $u \times \lambda \times n$, significantly lower than TSS. Since PP communication allows MRACS to use BS without significantly impacting player responsiveness, MRACS has far lower processing compared to MS. For example, Cronin *et al.* [43] found that every command and rollback takes $0.144ms$ and $1.41ms$, respectively; and the probability of a rollback occurring is 0.056. Assuming every player generates 20 updates per second, Figure 4.6 shows the processing time for 1 seconds worth of updates for MS using TSS with 4 states [43] and MRACS using BS. The figure shows that MS can only support up to 74 players, beyond which the mirrors are saturated (players are generating updates faster than they can

		C/S & RACS (PRP)	**RACS (PP)**	**MS & MRACS (PRP)**	**MRACS (PP)**
Player	**In**	$o+a\times\|MRP\|$	$a\times(o+\|MPP\|)$	$o+a\times\|MRP\|$	$a\times(o+\|MPP\|)$
	Out	$o+\|MPR\|$	$o+\|MPR\|+a\times(o+\|MPP\|)$	$o+\|MPR\|$	$o+\|MPR\|+a\times(o+\|MPP\|)$
Server / Mirror (Public)	**In**	$n\times(o+\|MPR\|)$	$n\times(o+\|MPR\|)$	$\frac{n}{m}\times(o+\|MPR\|)$	$\frac{n}{m}\times(o+\|MPR\|)$
	Out	$n\times(o+a\times\|MRP\|)$	*negligible*	$\frac{n}{m}\times(o+a\times\|MRP\|)$	*negligible*
Mirror (Private)	**In**	**N/A**	**N/A**	$m\times(o+\frac{n}{m}\times\|MRR\|)$	$m\times(o+\frac{n}{m}\times\|MRR\|)$
	Out	**N/A**	**N/A**	$o+\frac{n}{m}\times\|MRR\|$	$o+\frac{n}{m}\times\|MRR\|$

Table 4.2: Bandwidth analysis of various architectures.

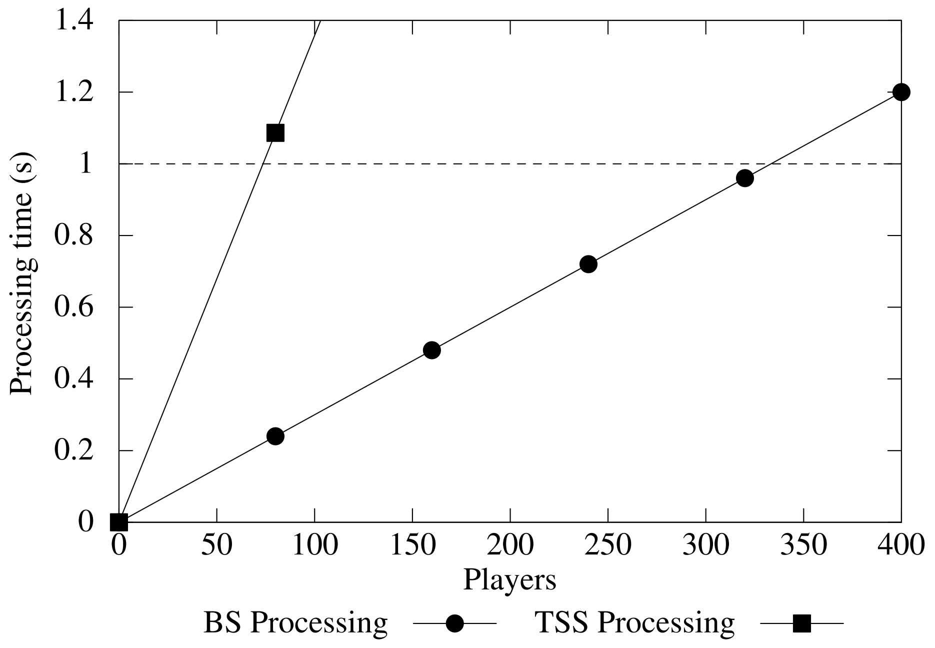
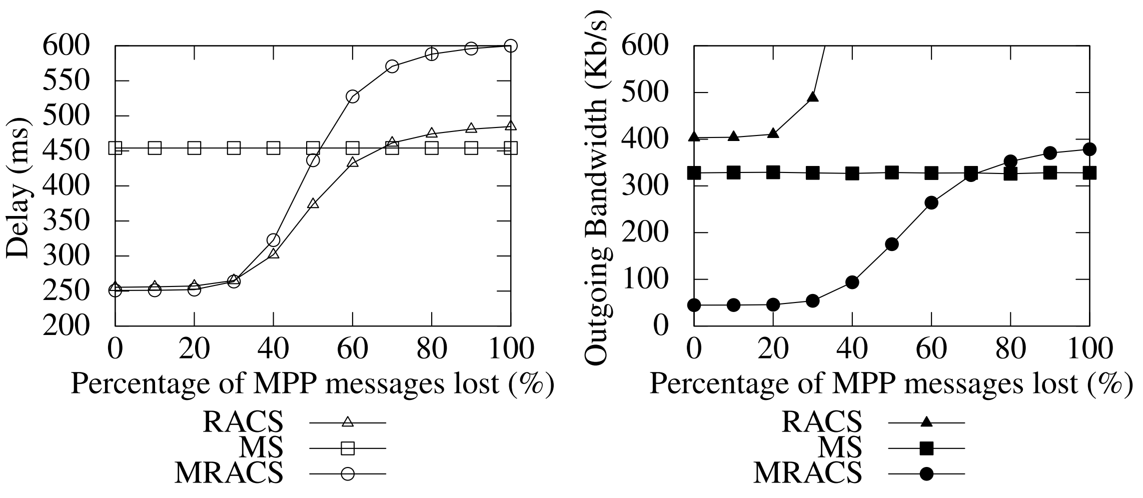


Figure 4.6: BS and TSS processing time.

be processed); whereas MRACS can support up to 333 players. MRACS with PP mode further reduces the mirrors' processing requirements as fewer updates are sent.

4.3.2 Bandwidth, Processing, and Delay - Simulation Results

To evaluate MRACS against RACS and MS we simulated all three using the Network Game Simulator (*NGS*) [138]. To increase scalability most MMOG divide the virtual world into zones, each with a fixed maximum capacity. For example, *World of Warcraft* (*WoW*) limits each zone to 300 players [110]. Players cannot enter a zone at full capacity, and players in different zones cannot interact. Therefore, the bandwidth and processing requirements grow linearly with the number of zones. Similar to Section 3.4.3.2, we simulate a world comprising a single zone of size 1000×1000 units, containing 300 players each controlling an avatar with an AoI radius of 100 units. Avatar movement is controlled using the Random-Way-Point (RWP) mobility model, with a velocity of two units/s and a wait time of zero. Note that the RWP model represents the best case for MS, RACS, and MRACS, but the worst case for showing the benefit of MRACS over RACS and MS. Using a more realistic mobility model would cause a far greater bandwidth increase in MS and RACS than MRACS. We simulated 1000 seconds with round length $\tau = 250ms$. To maximise responsiveness rounds begin every $50ms$ (rounds are pipelined); hence, clients generate 20 updates per second. As this thesis does not address the mirror placement problem we use an artificial network topology. For MRACS and MS the network topology is similar to that in Figure 4.1, with fully connected mirrors and the private network delay fixed at $50ms$ [43]. As communication over the Internet will be considerably slower than the private network, the player-mirror delay is $200ms$, while the player-player delay is $250ms$. The player-player



(a) Game state delay.

(b) Mirror/referee out-bandwidth.

Figure 4.7: RACS, MS and MRACS delay and out-bandwidth.

delay is larger than the player-mirror delay as each player connects to its closest mirror. For RACS we assume the player-referee delay is $250ms$, as there is only one referee serving all clients.

Simulation 4.1 compares MRACS against RACS and MS in terms of bandwidth and delay with 10 mirrors for MRACS and MS. Following Cronin *et al.* [43], we considered MS using TSS with a leading state of $0ms$ and trailing states $50ms$, $100ms$, and $150ms$ with a rollback probability of 0.044, 0.006, and 0.006, respectively. We considered MRACS using BS with $\Delta = 150ms$, so that MS and MRACS have equal consistency and worst case delay. For MRACS and RACS, we set $w = 6$, $s = 200$, and $p = 94\%$, as these settings are appropriate for fast paced Internet games (See Section 3.4.3.4. These settings assume the client software can interpolate/extrapolate up to 6 consecutive lost updates and that dropping fewer than 12 updates every ten seconds (94% of the previous 200 messages were received) will give a cheater an insignificant advantage. If a pair of players reverts to PRP mode, they will not re-attempt PP mode for 60 seconds.

Figures 4.7(a) and 4.7(b) show the players' average game state delay and mirror/referee out-bandwidth, respectively, with increasing packet loss between players due to network loss, cheating, or firewalls. MS has nearly fixed delay and bandwidth; however, the delay and bandwidth are high as all updates are routed through the mirrors. On the other hand, with 0% loss (*i.e.*, loss-less network and no cheaters) all players in MRACS are in PP mode and therefore very few messages are routed through the mirrors; hence, the bandwidth and delay in MRACS is far lower than MS. The bandwidth used by the single referee in RACS is far higher than MS and MRACS as it must receive updates from all players. As packet loss increases, players using MRACS and RACS revert to PRP communication, increasing the mirrors'/referee's bandwidth and players' average delay. As all communication in RACS is performed by a single referee, the bandwidth cost increases rapidly; hence, RACS has poor

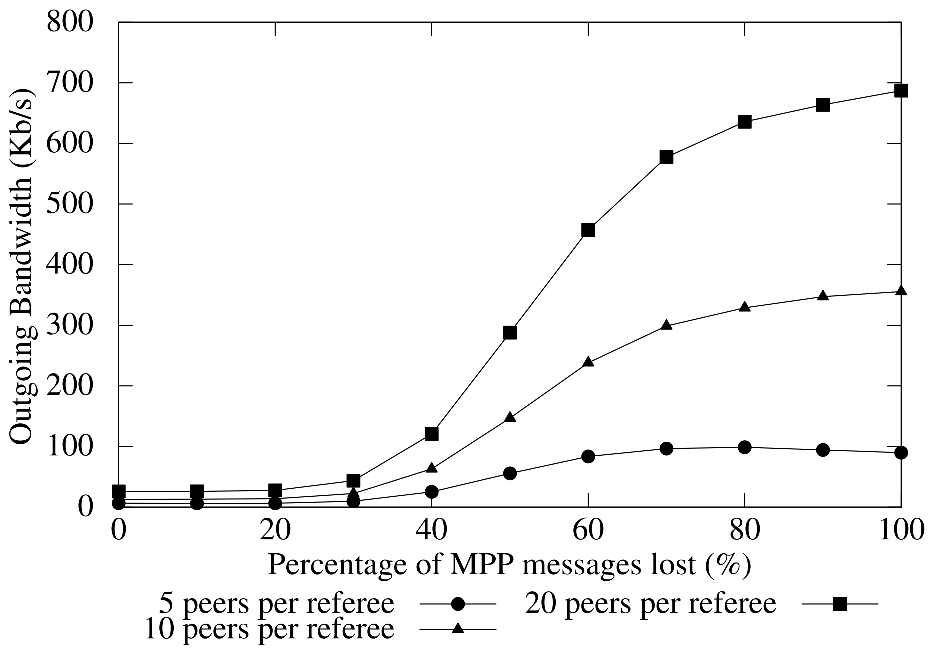


Figure 4.8: MRACS mirror out-bandwidth.

bandwidth scalability when more than 30% packet loss occurs. For MRACS, when packet loss exceeds 40%, players rapidly revert to PRP communication, dramatically increasing the impact of BS, increasing client delay. Above 60% packet loss MRACS has higher delay than MS due to the delay introduced by BS. Further, above 70% packet loss the bandwidth of MRACS exceeds MS due to the cost of sending digital signatures, hashes *etc.*, which are required to achieve cheat-proofing equivalent to C/S, exceeding that of MS. As Internet loss rates are typically less than 1% [38] and assuming less than $60 - 1 = 59\%$ of players are using protocol cheats or are firewalled from peers, both solvable by the players themselves, MRACS outperforms RACS and MS using this topology.

Simulation 4.2 evaluates the bandwidth scalability of the mirrors in MRACS with 300 players. We repeated Simulation 4.1 using 20, 10, and 5 mirrors, each supporting 15, 30, and 60 players, respectively. As shown in Figure 4.8, MRACS offers excellent bandwidth scalability when most players use PP mode (*i.e.*, with global loss rates below 40%); the mirrors' bandwidth is minimal, as they do not forward updates. However, the bandwidth increases as the number of PRP players increases. Figure 4.8 also shows that the bandwidth requirements of MRACS are proportional to the number of players per mirror; therefore, more mirrors increases available bandwidth and distributes it throughout the network.

Figure 4.6 shows that BS has far lower processing overhead than TSS, but does not show its impact on responsiveness. Provided the majority of players use PP mode BS does not add significant delay. However, a high MPP loss rate due to cheating or poor connectivity will cause the majority of players to revert to PRP mode, resulting in reduced responsiveness caused by BS. For comparison we simulated MRACS using TSS and MRACS using BS; all other parameters are identical to Simulation 4.1. As shown in Figure 4.9, TSS

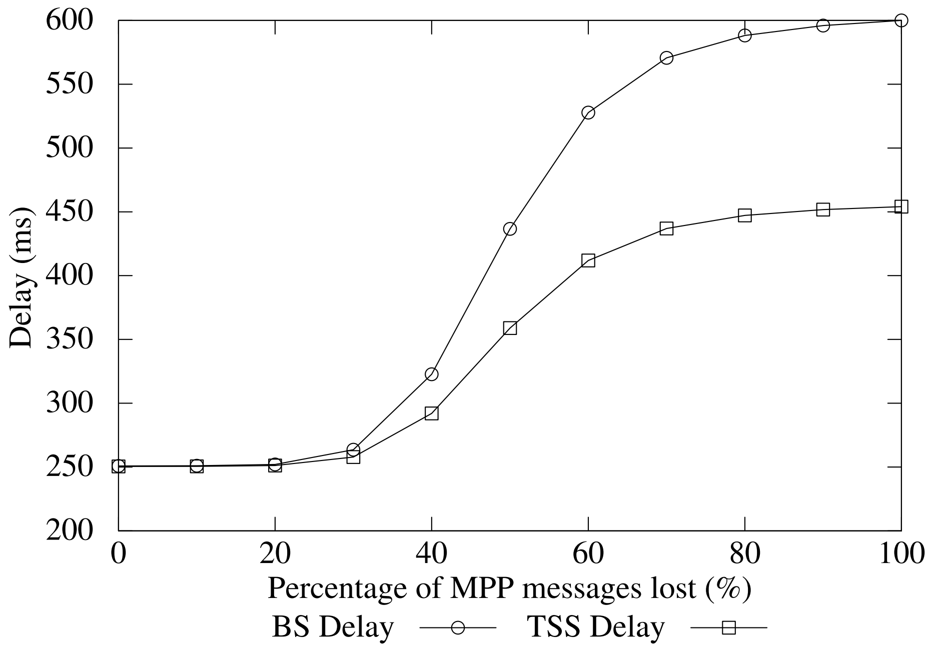


Figure 4.9: Delay for MS using TSS and MRACS using BS.

does provide lower delay, but only when the percentage of lost MPP messages exceeds 30%. When the loss rate is below 30%, BS is superior to TSS for MRACS due to its lower processing overhead.

4.3.3 CMA Simulation Results

The efficiency and effectiveness of CMA algorithms are influenced by the player and mirror locations and by the player to mirror delays. We used the two realistic simulation inputs described in Section 3.4.3.1 to simulate two scenarios; one simple and one realistic, described in Section 4.3.3.1 and Section 4.3.3.2 respectively. Note that the connected player population of an MMOG is centred in the time zone experiencing evening [110]. By using a single *Counter-Strike* trace, and as players gravitate to servers with low delay [56], our simulation inputs approximate this phenomenon. For both scenarios, we selected 10 IP addresses to act as mirrors according to the distribution of *Counter-Strike* servers [56]: 4 in the United States (San Francisco, Seattle, Dallas, and New York), 4 in Europe (Berlin, Stuttgart, Paris, and London), 1 in Asia (Singapore), and 1 in Australia (Sydney).

4.3.3.1 Simple Scenario

We compared J-Greedy/L-Ignore, J-Greedy/L-Greedy, and J-SA/L-SA in two phases: join (players join in random order until the system is full) and leave (players leave in the order they joined until the system is empty). The system was initially empty, and players join/leave in identical order for each algorithm. This scenario demonstrates the behaviour of each algorithm clearly. Each mirror M_j has approximately the same capacity ($C_j \approx 720$).

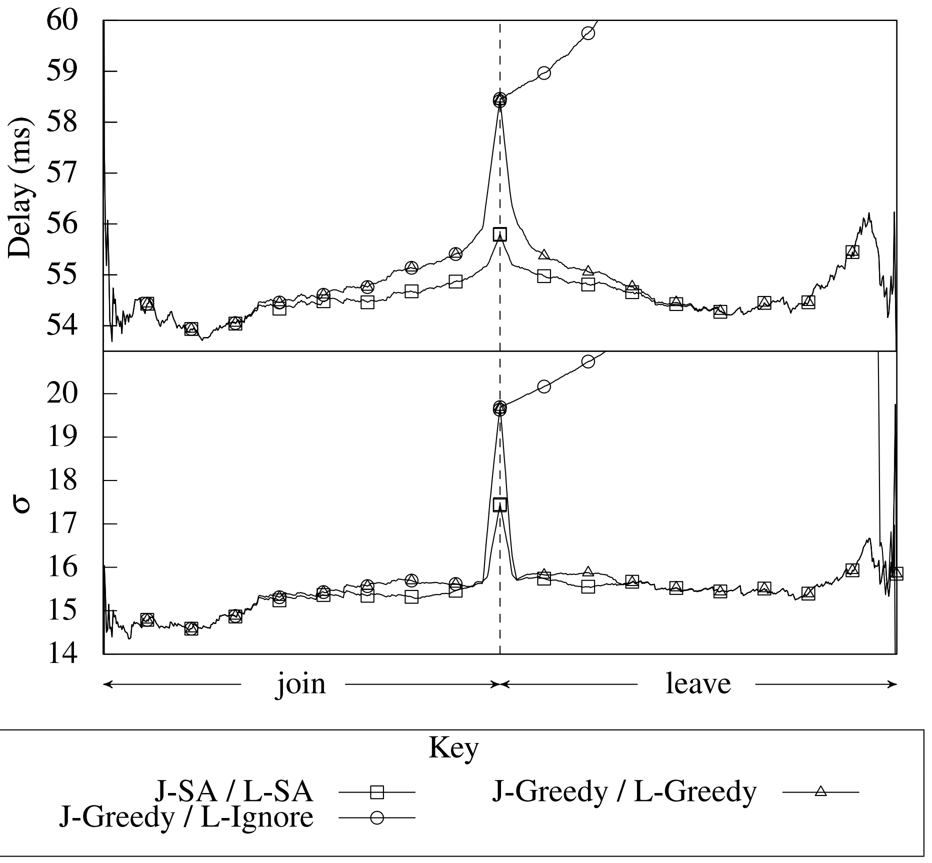


Figure 4.10: Average delay and standard deviation for the simple case.

After each join/leave the average, min, max, and standard deviation (σ) of players' delays were calculated. As shown in Figure 4.10, J-Greedy produces optimal results when no mirror has reached its capacity. However, with one or more saturated mirrors, J-Greedy is not optimal; the closer the system is to full capacity the worse the performance of J-Greedy. This is evident from the sharp increase in average delay and standard deviation when the system exceeds 95% capacity. The difference in average delay for J-Greedy and J-SA at 100% capacity is 2.65ms, which seems insignificant because it is averaged across 7207 players. To highlight the impact of each algorithm on individual players, Figure 4.11 plots the difference in delay for each player between J-Greedy and J-SA after all players have joined; a positive value indicates the player has lower response time with J-SA than J-Greedy, and vice-versa. Figure 4.11 shows the delay difference between J-SA and J-Greedy when the system is at 100% capacity. A positive value indicates the player has lower delay with J-SA than J-Greedy and vice-versa. The figure shows that when the system exceeds 95% capacity joining players receive very poor assignments using J-Greedy, up to 87 ms slower than J-SA; while the reassigned players using J-SA receive a penalty of at most 24ms; therefore, the benefit outweighs the cost. These results show that J-Greedy produces good results when the system has spare capacity; however, J-SA is far better when the system is near full capacity. Note, the longest possible chain of reassignments for J-SA is $m-1$; however, the average and maximum chain lengths in the simulation were only 0.55

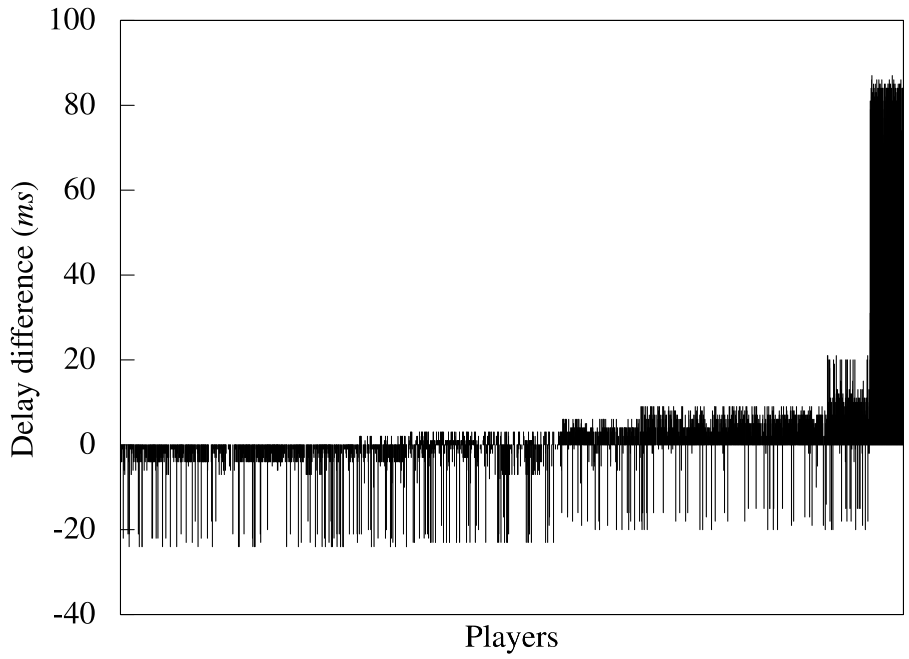


Figure 4.11: Delay difference for simple case.

and 6 respectively.

For the leave phase, Figure 4.10 shows that L-Greedy rapidly improves the assignments of players when the system capacity is below 95%. Except when the system is close to full, J-Greedy is close to the optimal L-SA. However, L-Ignore results in significantly worse delay as it does not improve the assignment of players. As in J-SA, the longest possible chain of reassignments for L-SA is $m-1$; however, the average and maximum chain lengths in the simulation were only 0.31 and 6 respectively.

Note, the min and max player delays are not shown in Figure 4.10 as they are approximately equal for all algorithms (min $\approx 40ms$, max$\approx 130ms$). This indicates that for each algorithm at least one player is assigned to a mirror in his city, and an assignment algorithm cannot improve the delay of players in distant parts of the network.

4.3.3.2 Realistic Scenario

In reality player joins and leaves are interleaved. Pittman and GauthierDickey [110] found that the rate of player joins for a *WoW* realm peaked at 500 players per hour. Further, most player sessions last less than 200 minutes, with a mean of 80 minutes, and can be accurately modelled using the Weibull distribution [110]. Using these as inputs, for each of the 7207 players to join and leave exactly once required simulating 34 hours. The number of active players approximately stabilised after 4 hours, and the maximum concurrent player population was 737. After 14.4 hours all players had joined. We used 10 mirrors, each with capacity for 74 clients. As in the simple simulation, statistics were calculated after each join/leave. Figure 4.12 shows the average delay and standard deviation when the system capacity is approximately stable. When the player population is below 95% system

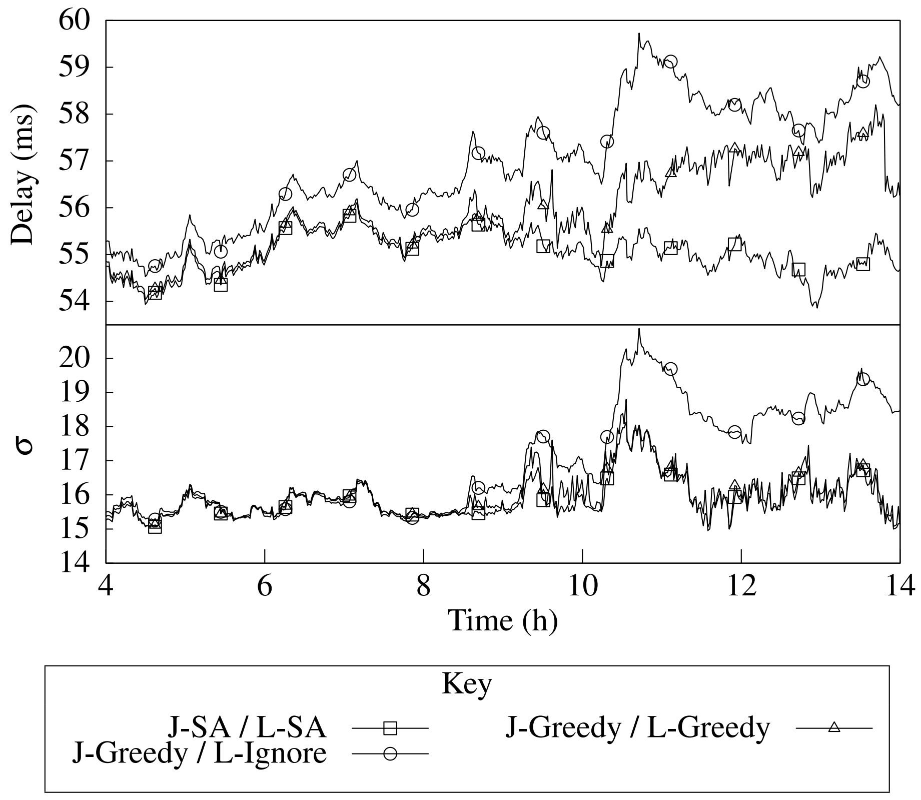


Figure 4.12: Average delay and standard deviation for the realistic case.

capacity (hours 4-10), J-Greedy/L-Greedy is far closer to the optimal delay (J-SA/L-SA) than that obtained by J-Greedy/L-Ignore. However, when the system capacity exceeds 95% (hours 10-14) joining players suffer poor assignments, significantly increasing the average delay and standard deviation for J-Greedy/L-Greedy. As in the simple simulation, the min and max delays (not shown) are approximately equal for all algorithms. The average and maximum chain lengths for J-SA/L-SA were only 1.3 and 7 respectively.

To measure the impact on the delay of individual players due to client reassignments when using J-SA/L-SA, the delay for every assignment of each client was recorded. The client with the highest delay range, $26ms$, had 12 assignments, max delay $98ms$, mean delay $74ms$, and standard deviation $7.15ms$. Due to the low delay range, max delay, and standard deviation, we believe the player's performance would not be significantly impacted by mirror reassignments.

4.3.3.3 Speed Comparisons

Table 4.3 compares the average time of each of the CMA-J and CMA-L algorithms for the simple and realistic scenarios, averaged across 10 input sets, each averaged across 100 runs. The entries for 10 mirrors correspond to the results in Sections 4.3.3.1 and 4.3.3.2. Consistent with their time complexities, J-SA is approximately 29% slower than J-Greedy, and L-SA is approximately 18% slower than L-Greedy. In the realistic scenario, however, J-SA/L-SA is only 8% slower than J-Greedy/L-Greedy. We repeated the simulations with

	Average join/leave time (*ms*)			
	m=5	***m=10***	***m=20***	***m=40***
Simple Scenario				
J-Greedy	0.0069	0.0114	0.0246	0.0578
J-SA	0.0075	0.0147	0.0356	0.0966
L-Ignore	0.0033	0.0042	0.0063	0.0082
L-Greedy	0.0047	0.0067	0.0135	0.0171
L-SA	0.0046	0.0079	0.0245	0.0438
Realistic Scenario				
J-Greedy/L-Ignore	0.0757	0.0810	0.0937	0.1272
J-Greedy/L-Greedy	0.0776	0.0830	0.0996	0.1374
J-SA/L-SA	0.0806	0.0894	0.1392	0.2713

Table 4.3: Average algorithm running time.

5, 20, and 40 mirrors with 7212, 7197, and 7177 players, respectively. The results show that J-Greedy and L-Greedy scale better than J-SA and L-SA, being approximately twice as fast with 40 mirrors. However, even with 40 mirrors, the maximum time for J-SA and L-SA was only 35*ms* and 25*ms*, respectively, which would not be noticed by a player. Therefore, if the system is expected to be close to capacity, we recommend using J-SA and L-SA.

4.4 Summary

In this chapter, we have proposed the MRACS architecture and showed that it performs better than the RACS and MS architectures. MRACS uses multiple mirrored referees to reduce the referee's bandwidth bottleneck in RACS, reduce the player-to-referee delay (increasing responsiveness and fairness), and remove the single point of failure. In contrast to MS, MRACS allows players to directly exchange updates (increasing the mirrors' bandwidth scalability, responsiveness, and fairness), uses RACS's cheat prevention protocols, and utilises a more efficient synchronisation algorithm (increasing processing scalability). MRACS considers dynamic player joins and leaves, which necessitates the use of a more effective CMA algorithm than that in MS, which assigns each client to its closest mirror. We have defined the CMA problem and proposed two pairs of algorithms to solve it: the optimal J-SA/L-SA, and the faster heuristic J-Greedy/L-Greedy. Our simulations show that the optimal and heuristic algorithms produce significantly lower client delays than J-Greedy/L-Ignore. The J-SA/L-SA solution performs better than J-Greedy/L-Greedy when the system is above 95% capacity; otherwise, the latter scheme, which is significantly faster, produces near optimal results. In Chapter 5 we investigate distributing referees to untrusted player hosts to minimise the required publisher/developer infrastructure.

Chapter 5

The Distributed Referee Anti-Cheat Scheme (DRACS)

The RACS and MRACS architectures have superior scalability, responsiveness, and fairness than C/S and MS respectively. However, the publisher must provision the infrastructure for running the referee(s). In particular, provisioning servers to host the referee(s) is expensive, and the bandwidth required for running them is an expensive recurring cost [100]. Further, although RACS and MRACS have better scalability than C/S and MS, they must still receive and simulate all player updates. Thus, a rapid influx of new players still has the potential to overload the system. Finally, while the peer-to-referee delay in RACS and MRACS is less critical than in C/S or MS, low peer-to-referee delay is still beneficial as the referee's game state is authoritative. Therefore, peers in distant parts of the network are disadvantaged.

To maximise responsiveness, fairness, and scalability, and minimise the publisher provisioned infrastructure, this chapter proposes the Distributed Referee Anti-Cheat Scheme (DRACS) in which referees are distributed to peers. Note that DRACS is a P2P architecture; thus its scalability is intrinsic to that of the P2P system. Therefore, this chapter does not address the scalability evaluation of DRACS. While the discussion in this chapter is focused on RACS, the principles are applicable to other P2P network game architectures, *e.g.,* [36, 64, 76]. DRACS has the following benefits over RACS and MRACS:

1. The publisher provisioned infrastructure is minimised, reducing costs.
2. The referee can be dynamically located close (in terms of delay) to interacting peers, maximising responsiveness.
3. The referee can be dynamically located with even delay to interacting peers, maximising fairness.
4. DRACS is resource growing, making it highly scalable. Each joining player is a potential candidate for hosting a referee.

5. The pool of candidate referees naturally follows the diurnal movement of the player population; thus, avoiding the server placement problem of C/S, RACS, MS, and MRACS.

While distributing referees to peers has significant advantages, there are several disadvantages that must be overcome:

1. The publisher loses control of the game state, creating additional opportunities for cheaters.
2. The peers' bandwidth and processing requirements increase.
3. As the reliability of individual peers is far below that of a well provisioned server, the protocol must include a fault tolerant mechanism to recover from multiple simultaneous referee failures.
4. As a single peer does not have sufficient resources to maintain the entire game state it must be divided among multiple peers. Further, a mechanism must exist such that each peer can access the appropriate portion of the authoritative game state, *i.e.*, achieving global connectivity.

This chapter investigates referee selection such that responsiveness and/or fairness are maximised (benefits 2 and 3), while addressing the potential for cheating (disadvantage 1). We assume peers have sufficient capacity to support a referee (*i.e.,* we do not address disadvantage 2). However, the massive success of P2P networks [99] shows that most desktop users will have sufficient capacity; low bandwidth devices such as mobile phones may not. Protocols to provide reliability, distribute the game state, and maintain global connectivity (disadvantages 3 and 4) are tangential to these problems; thus, they can be addressed by future work.

The layout of this chapter is as follows. Section 5.1 describes the DRACS architecture. Sections 5.2, 5.3, and 5.4 describe our system model, propose the referee selection problem, and propose two possible solutions, respectively. Section 5.5 uses simulation to evaluate our proposed selection algorithms. Finally, Section 5.6 summarises the chapter. Note, the work in this chapter was originally published in [146], and was nominated for the best paper award; therefore, an extended version was published in SCS Simulation [147]. Related work on the accuracy of delay estimation schemes was published in [143].

5.1 Distributed Referee Anti-Cheat Scheme

As shown in Figure 5.1, DRACS comprises three entities: an authentication server (S_A), a set of n players $P = \{P_i \mid P_i$ is a player with unique ID $i\}$, and a set of m referees $R = \{R_f \mid R_f$ is a referee with unique ID $f\} \subseteq P$. DRACS assumes the virtual world is divided into

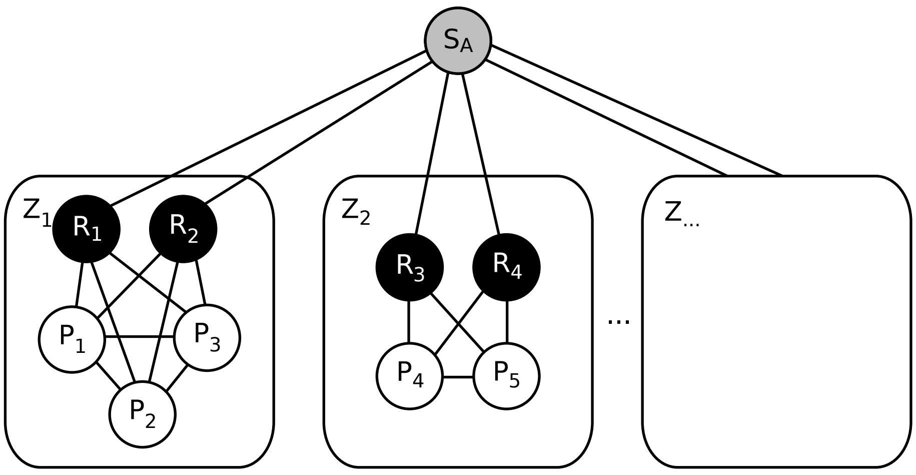


Figure 5.1: DRACS architecture.

discrete regions called zones [30, 58, 92, 110, 154], either dynamically or statically, and that a mechanism exists to transfer players between zones. Let Z_x denote a zone x, $Z_x^P \subseteq P$ be the set of players located within zone Z_x, and Z_x^R be the set of referees controlling zone Z_x. Each player perceives only a small portion of the virtual world and AoI filtering is used to reduce the size of updates; zones are considerably larger than a player's AoI.

DRACS uses the principle of mutual checking *"you may not trust a single client, but you trust the consensus of multiple unaffiliated clients*" [77] to prevent a single player cheating. Further, as mutual checking requires referees to be unaffiliated, it also prevents groups of colluding players from cheating. Note that this solution also prevents groups of griefers (described in Section 2.2.3) from damaging the authoritative game state.

Referees form a consensus by voting on the authoritative game state. To minimise voting delay DRACS uses the method proposed by Kabus *et al.* [76], in which each peer sends its updates to every referee, each referee simulates all peers' updates, and returns the result to the peers (a vote); the authoritative state is decided by the majority vote. To minimise network delay requires selecting referees with low peer-to-referee delay, without selecting multiple affiliated referees.

5.2 System Model

Figure 5.2 illustrates the relationships among players, referees, and colluding cheaters. As in Corman *et al.* [37], we distinguish between cheating players from colluding players (cheaters that work together to disrupt the game). Further, we assume the number of cheaters far exceeds the number of colluding players. Let $C = \{C_i \mid C_i$ is a set of colluding players$\}$. We assume there is no method for cheating players to identify each other (unless they are already colluding) as this same method could be used by the publisher to detect them [37]. Thus, all sets of colluders are disjoint, *i.e.,* $C_i \cap C_j = \emptyset$ for all $i \neq j$. The size of the largest group of colluding peers, $max(|C_i|)$, and the percentage of cheaters, K, is

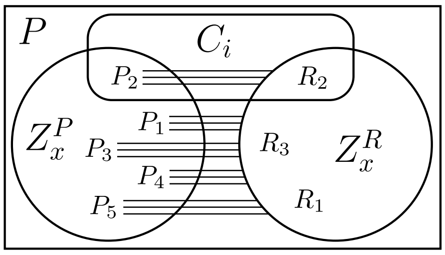


Figure 5.2: Peer membership.

unknown; however, we assume that the developer can estimate $maxC \approx max(|C_i|)$ and the percentage of cheaters $k \approx K$.

The publisher provisioned S_A is responsible for authenticating and validating joining players (subscription, banning, *etc.*), and selecting peers to act as referees; therefore, the referee selection process is trusted. Further, the publisher may provision a small number of dedicated trusted referees that can boot-strap the system when the number of players is small. Note, acting as a referee requires additional bandwidth and processing requirements; therefore, only peers with sufficient resources should be considered for referees. The bandwidth and processing resources for each peer should be transmitted to the S_A as part of the authentication process to achieve this.

Each peer can play and referee at the same time; therefore, $Z_x^R \subseteq P$. A referee should not control the zone in which his avatar is located. In other words, each R_f in Z_x^R is not in Z_x^P, *i.e.*, $Z_x^R \cap Z_x^P = \emptyset$. If a referee's avatar moves into a zone it is controlling, then the S_A will select a new referee as a replacement. The following additional checks may be performed to improve security: (i) a referee must not share the IP address of a player whose avatar is located within the zone (multiple players sharing one Internet connection), and/or (ii) a referee must not control a zone containing players where game mechanics indicate bias (*e.g.*, if they are both members of the same team/guild/clan).

The number of referees per zone, $|Z^R|$, should be set by the developer. Each player in Z_x^P receives the game state from all R_f in Z_x^R (three lines joining each P_i to Z_x^R in Figure 5.2, $|Z^R| = 3$) and takes the majority result; therefore, it requires at least $\left\lceil \frac{|Z^R|}{2} \right\rceil$ colluding referees controlling one zone to tamper with the game state.

Finally, we model game fairness as the range of the average delay for each peer in Z_x^P to all referees in Z_x^R. If all peers receive updates simultaneously from the referees, then the game is completely fair (delay range 0), whereas, the higher the range of delays, the greater the unfairness. Note that a game with very high delay may be fair, but unplayable. To be fun a game should be both fair and playable.

5.3 Referee Selection Problem

Let $d_{i,f}$ be the delay from a player P_i to / from a referee R_f; we assume symmetric delay, *i.e.*, $d_{i,f} = d_{f,i}$. Given $|Z^R|$ and Z_x^P for zone Z_x, and the publisher's pre-defined $0 \leq S_{max} \leq 1$

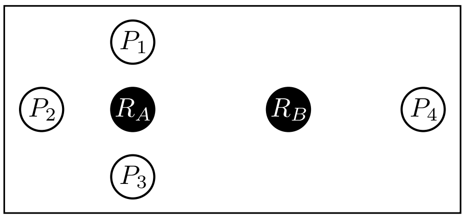


Figure 5.3: Referee selection example.

and *maxC*, the Referee Selection Problem (RSP) is to select a referee set Z_x^R such that:

1. the probability S_x that there are $\left\lceil \frac{|Z^R|}{2} \right\rceil$ or more colluding referees in Z_x^R is not larger than S_{max};
2. the average peer-to-referee delay, $\overline{d_{Z_x^P, Z_x^R}} = \frac{\sum_{P_i \in Z_x^P, R_f \in Z_x^R} d_{i,f}}{|Z_x^P| \times |Z_x^R|}$, is minimised; and
3. the difference, Δ, between the maximum and the minimum of the player-referee delays (averaged across all referees in Z_x^R) for all peers in Z_x^P is minimised, *i.e.*, minimise $\Delta = max(\overline{d_{i,Z_x^R}}) - min(\overline{d_{j,Z_x^R}})$, where $\overline{d_{i,Z_x^R}} = \frac{\sum_{R_f \in Z_x^R} d_{i,f}}{|Z_x^R|}$ is the average delay from a player P_i in Z_x^P to all referees in Z_x^R.

The game developer should set the probability S_{max} based on the security requirements of the game; lower S_{max} improves security, as it reduces the chance that the majority of referees in Z_x^R are colluding. Further, recall that colluders in one group cannot locate members of another group (see Section 5.2); therefore, even if every selected referee is a cheater, provided the majority of referees are from different groups of colluders, security is maintained. Hence, the RSP considers security only against groups of up to *maxC* colluders, not the union of all groups of colluders. Note that if $max(|C_i|) > maxC$, or $K > k$, then we may not be able to select Z_x^R that meets $S_x \leq S_{max}$.

Criteria 2 and 3 are used to improve the game's QoS; criterion 2 deals with improving game responsiveness, while criterion 3 addresses game fairness. Since a valid game state is decided by the majority of referees (not the consensus of all referees), using the average value (not the maximum delay) in criterion 3 is sufficient to achieve fairness. To illustrate the criteria, consider Figure 5.3 that includes $Z_x^P = \{P_1, P_2, P_3, P_4\}$ and assume we want to select one referee from two candidate referees, R_A and R_B. Consider the following player-to-referee delays (in *ms*): $d_{1,A} = 10$, $d_{1,B} = 40$, $d_{2,A} = 20$, $d_{2,B} = 60$, $d_{3,A} = 10$, $d_{3,B} = 40$, $d_{4,A} = 100$, $d_{4,B} = 60$. If R_A is selected, $\overline{d_{i,f}} = 35$ and $\Delta = 90$. However, selecting R_B will give $\overline{d_{i,f}} = 50$ and $\Delta = 20$, which is better in terms of criterion 3 but worse for criterion 2. Note that selecting the peers closest to the players in Z_x^P as referees will obviously optimise criterion 2, but could compromise criterion 1 as it is probable that colluding peers will be located within the same part of the network. In general, as all three criteria are conflicting it is not possible to simultaneously optimise all of them.

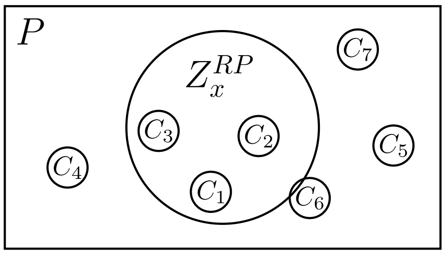


Figure 5.4: Colluding peer membership.

5.4 Secure Referee Selection Algorithms

The authentication server, S_A, holds responsibility for selecting referees. It will run one of the referee selection algorithms, described in Section 5.4.3 and 5.4.4, for each zone of the game world.

5.4.1 Estimating Delay between Peers

To address criteria 2 and 3, a solution to the RSP requires knowing all peer-to-peer delays. Each $d_{i,j}$ can be measured using echo packets between peers i and j. One could keep the delays in a $|P| \times |P|$ delay matrix since any peer may potentially act as a referee. Creating this matrix requires $O(|P|^2)$ measurements and space, which becomes infeasible for large $|P|$; the time cost is even worse if we consider maintaining the matrix given the dynamic nature of players and the Internet.

We propose the use of network coordinates [44] to estimate peer-to-peer delay. Network coordinates provide a good estimation of the delay between any two peers i and j, even if no direct measurements between i and j have been made. In contrast to using a delay matrix, in this approach we estimate delay only when it is needed. Therefore, it is much more bandwidth and space efficient. Note that one can use other methods (*e.g.*, landmarks [60, 102] or geographic location [93]) to estimate peer delays. We assume that each peer calculates its own network coordinate and transmits it to the S_A.

5.4.2 Size of the Candidate Referee Set

Let $Z_x^{RP} = P - Z_x^P$ be the referee pool from which each R_f in Z_x^R is selected. As shown in Figure 5.4, Z_x^{RP} may include players from more than one C_i.

The probability S_x of selecting at least $\alpha = \left\lceil \frac{|Z^R|}{2} \right\rceil$ colluding referees is given by

$$S_x = \frac{\sum_{i=\alpha}^{|Z^R|} \left\lceil \frac{k\psi}{maxC} \right\rceil \binom{maxC}{i} \binom{\psi - maxC}{|Z^R| - i}}{\binom{\psi}{|Z^R|}} \tag{5.1}$$

where ψ is the size of the candidate referee set, $Z_x^{CR} \subseteq Z_x^{RP}$. We want to find the minimum value of ψ such that randomly selecting $|Z^R|$ peers from Z_x^{CR} will limit S_x to no more than

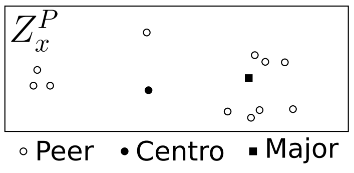


Figure 5.5: Example 2D network coordinates.

S_{max}. A simple brute force approach suffices to find this minimum value, since the values of all other variables in Equation (5.1) are known.

Randomly selecting Z_x^{CR} from Z_x^{RP}, as in Corman *et al.* [37], will likely result in obtaining Z_x^R with large peer-to-referee delays and a large range of delays, which fails to satisfy criteria 2 and 3. In the following subsections, we propose two algorithms, SRS-1 and SRS-2, that meet criterion 1 and balance criteria 2 and 3.

5.4.3 SRS-1

Algorithm SRS-1 emphasises responsiveness (criterion 2) while satisfying security (criterion 1). Given the pre-computed minimum size $\psi = |Z^{CR}|$ from Equation (5.1), SRS-1 performs three steps:

1. Select the candidate referee set $Z_x^{CR} \subseteq Z_x^{RP}$, such that selecting any subset $Z_x^R \subseteq Z_x^{CR}$ will give a small value of $\overline{d_{Z_x^P, Z_x^R}}$.
2. Select $|Z^R|$ referees randomly from Z_x^{CR}.
3. Artificially inflate peer delays to $d_{F\%}$ (defined later) such that $F\%$ of peers have equal delay.

For Step 1, SRS-1 selects referees close to the majority of players. Let (x_i, y_i) be the coordinates of P_i. Informally, we define the *major* as the point in the coordinate space that is closest to the majority of players in Z_x^P. As an illustration, consider Figure 5.5 that shows an example Z_x^P in 2D coordinate space. The total distance, $TD(x,y)$, from a coordinate (x,y) to all players is:

$$TD(x,y) = \sum_{P_i \in Z_x^P} dist((x,y),(x_i,y_i)) \tag{5.2}$$

where $dist(A,B)$ is the distance between points A and B. Formally, the *major* is the point that minimises Equation (5.2). SRS-1 will form Z_x^{CR} in Step 1 by selecting the ψ peers not in Z_x^P that are closest to the *major* coordinate. Thus, selecting referees close to the *major* addresses criterion 2. The random referee selection in Step 2 addresses criterion 1.

Since this selection does not consider the range of delays, a large range may result, so Step 3 addresses criterion 3. The mechanism to decrease the range of delays is that a referee can send updates late to a peer with low delay so that it receives updates at the same time as a peer with high delay, artificially giving these peers the same delay. We define the fairness weight $0 \leq F \leq 100\%$ in Step 3 as the minimum percentage of peers who must have equal average delay to the referees. Specifically, for each peer we calculate the average delay to all referees, find $d_{F\%}$ (the maximum delay among the fastest $F\%$ of players), and inflate the delay of the fastest $F\%$ of peers to $d_{F\%}$. The developer may set the weight of F between 0 and 100% to balance responsiveness and fairness (criteria 2 and 3). If SRS-1 inflated delay for all peers ($F = 100\%$), as proposed by Aggarwal *et al.* [6], it would inflate all peer delays to that of the slowest peer, possibly undermining the purpose of Step 1. Using SRS-1 with $F = 0\%$ is analogous to current commercial games [135] which attempt to provide the fastest service possible to each individual player, ignoring fairness. As an alternative to setting a fixed weight for F in Step 3, one may use outlier detection (*e.g.*, the box-plot method [133]) to ignore peers with very high delay when calculating the inflation value.

The referee selection algorithm SRS-1 is shown in Algorithm 5.1. The $major(Z_x^P)$ function returns the *major* for the set of peers Z_x^P, calculated as the median x and median y values of all players in Z_x^P. The find_CR((x, y), ψ, $P - Z_x^P$) function returns the ψ peers $P_i \notin Z_x^P$ closest to the *major* coordinates (x, y). Function slowest_peer(Z_x^P, Z_x^R, F) returns the average peer-to-referee delay of the $F \times |Z_x^P|^{\text{th}}$ slowest peer. Finally, function inflate_peers(Z_x^R, Z_x^P, $d_{F\%}$) notifies all referees in Z_x^R to artificially inflate the delay to peers in Z_x^P to $d_{F\%}$. Note that if Z_x^R is already partially populated the algorithm will select only the number of referees required to fill the zone. For example, the algorithm can select a replacement referee when one leaves the game.

Algorithm 5.1: SRS-1(ψ, r, F)

```
/* $\psi$ - $|Z_x^{CR}|$ */
/* $r$ - $|Z^R|$ */
/* $F$ - The fairness weight */
begin
    $(x,y)$ = major($Z_x^P$)
    $Z_x^{CR}$ = find_CR($(x,y)$, $\psi$, $P - Z_x^P$)
    while $|Z_x^R| < r$ do
        $R_i$ = random($Z_x^{CR}$)
        $Z_x^R = Z_x^R \cup R_i$
        $Z_x^{CR} = Z_x^{CR} - R_i$
    end
    $d_{F\%}$ = slowest_peer($Z_x^P$, $Z_x^R$, $F$)
    inflate_peers($Z_x^R$, $Z_x^P$, $d_{F\%}$)
end
```

5.4.4 SRS-2

Algorithm SRS-2 emphasises fairness (criterion 3) while satisfying security (criterion 1). As for SRS-1, SRS-2 comprises three main steps:

1. Select the candidate referee set $Z_x^{CR} \subseteq Z_x^{RP}$ such that selecting any subset $Z_x^R \subseteq Z_x^{CR}$, will incur a small inflation value $d_{F\%}$.
2. Select $|Z^R|$ referees randomly from Z_x^{CR}.
3. Artificially inflate peer delays to $d_{F\%}$ such that the closest $F\%$ of peers have equal delay.

Note that Steps 2 and 3 of SRS-1 and SRS-2 are identical.

We can directly address criterion 3 by selecting referees that minimise the range of peer delays. However, this approach would unlikely be able to optimise criterion 2, resulting in a larger $d_{F\%}$. Therefore, SRS-2 selects referees close to the centre of Z_x^P, selecting referees such that the average delay from all referees to the furthest player is minimised. Let the *centro* be the point in the coordinate space such that the maximum distance to all players in Z_x^P is minimised. Let the maximum distance, $MD(x,y)$, from a coordinate (x, y) to all players be:

$$MD(x,y) = max(\forall P_i \in Z_x^P, dist((x,y),(x_i,y_i))) \tag{5.3}$$

The *centro* (illustrated in Figure 5.5) is the point that minimises Equation (5.3). However, if there are some outlying peers located in distant parts of the network they will have a significant impact on the *centro*. To prevent this SRS-2 may use outlier detection to ignore distant peers when calculating the *centro*. For each peer the delay to all other peers is calculated, and box-plot outlier detection [133] is used to identify distant peers. Similar to SRS-1, SRS-2 populates Z_x^{CR} with the closest ψ peers, $P_i \notin Z_x^P$, to the *centro*.

As some players may be located very close to the *centro*, there may still be a significant delay range. As in SRS-1, to achieve criterion 3, in Step 3 the referees artificially inflate peer delays to $d_{F\%}$. Note that SRS-2 with outlier detection is not effective when $F = 100\%$ as it results in generating a large inflated delay, and hence reduces responsiveness. Therefore, we suggest using SRS-2 with outlier detection only for $F < 100\%$.

SRS-2 is shown in Figure 5.1 by replacing function *major*(Z_x^P) with *centro*(Z_x^P). In this thesis, we use gradient descent [9] in function *centro*() to find the coordinates.

5.5 Simulation and Discussion

We use simulation to compare the effectiveness of SRS-1 and SRS-2 in addressing criteria 2 and 3 against random referee selection, which is equivalent to SGA [37]. The simulations require knowing the peer-to-peer delays, $d_{i,j}$, and their avatar locations. As no trace data from a real MMOG is available, we synthesised three representative topologies. Simulation 5.1 uses a topology constructed from public information about *World of Warcraft* (*WoW*) [21] and measured Internet delays [93]. Simulation 5.2 uses an artificial topology to high-

Region (%)	**Peers**
China: 44	2200
USA: 25	
Boston: 25	313
Dallas: 33	413
LA: 28	350
Seattle: 14	175
Europe: 19	
UK: 46	437
Germany: 35	333
France: 16	152
Spain: 3	28
Other: 12	599

Table 5.1: Player distribution for Simulation 5.1

light the difference between SRS-1 and SRS-2. Simulation 5.3 uses the realistic topology from Section 3.4.3.1 to demonstrate SRS-1 and SRS-2 in a realistic simulation.

5.5.1 Simulation 5.1

Table 5.1 shows the percentage of *WoW* players located in each geographical region [3, 20, 148]; *WoW* is one of the most popular MMOG to-date, with over 11.5 million subscribers globally [20]. We assume the *Other* players are located in Australia as it has significant delay to other regions [93] and a moderate *WoW* player base [148].

As shown in the table, for Simulation 5.1, we generated a network game topology with 5000 peers distributed according to the *WoW* player distribution. We used reference [93] to approximate the delay between these regions, and as most game players have broadband access [136], we added a last hop delay of $20ms$ [86] to all peers. We used the Vivaldi [44] simulator in matrix mode for 3×10^6 rounds to construct 2D network coordinates for all peers from the topology.

Following *WoW* that allows groups of up to 40 players [148], our simulation populates Z_x^P with 40 peers and selects $|Z^R| = 3$ referees using random referee selection (SGA), SRS-1, and SRS-2. To show the impact of the distribution of players in Z_x^P on each algorithm, we generated 41 different player distributions for $0 \leq W \leq 40$, where W is the minimum number of players located in the US. For each W value, we selected the other $40 - W$ players randomly from around the world (including the US). We assumed $k = 2\%$ [100], $maxC = 10$, and $S_x = 0.1$; therefore, $\psi = 50$. The experiment was repeated 100 times for each of the 41 player distributions and the results were averaged, for inflated values $d_{100\%}$, $d_{80\%}$, and $d_{60\%}$. To evaluate the performance of our solutions when delay is not inflated, the figure includes the average peer-to-referee delay. Note that the current industry standard attempts to maximise responsiveness for every player individually, ignoring fairness, and therefore the average delay measure reflects the current standard.

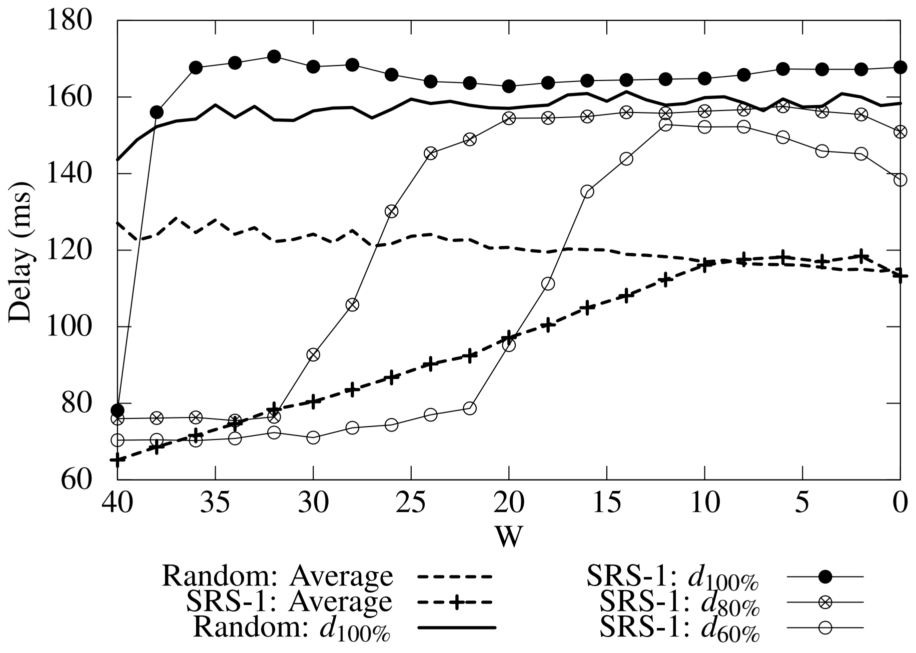


Figure 5.6: Simulation 5.1 results.

As shown in Figure 5.6, the average delay for SRS-1 outperforms random selection, even when only 25% of players are in the US ($W = 10$). As interacting players for an MMOG are often located in the same region of the network [33] (*e.g.*, above 50% in the US), SRS-1 should be very effective in practise.

The figure also shows the performances of the algorithms when the publisher sets the weight of F. As shown in the figure, for $F = 100\%$, SRS-1 is better than random selection only when $W \in [40, 37]$ (*i.e.,* at least 92.5% of the players are in the US). This result shows that it is not possible to achieve fairness and responsiveness to all players when even a small number of peers are located in distant parts of the network. Decreasing F trades fairness for responsiveness. The figure shows that when $F = 80\%$, the maximum delay remains the same even when $\frac{8}{40} = 20\%$ of the players are not in the US. Reducing F to 60% further reduces the effects of distant players (*i.e.*, tolerating almost $\frac{20}{40} = 50\%$ of peers outside the US) on the maximum response time, shifting the curve in the figure to the right. The results for SRS-2 for this topology are comparable to SRS-1, and therefore are excluded from the figure.

5.5.2 Simulation 5.2

The difference between SRS-1 and SRS-2 is not apparent from Simulation 5.1 due to the structure of the topology. For Simulation 5.2 we generated an artificial topology with three locations, East Coast (East), Central, and West Coast (West) of the US, with delays between East/West to Central of 50*ms*, and East to West of 100*ms*. From Table 5.1, 25% of players are located in the US, and therefore we generated $|P| = 25\% \times 5000 = 1250$ players. We distributed them evenly among the three locations, and generated network coordinates sim-

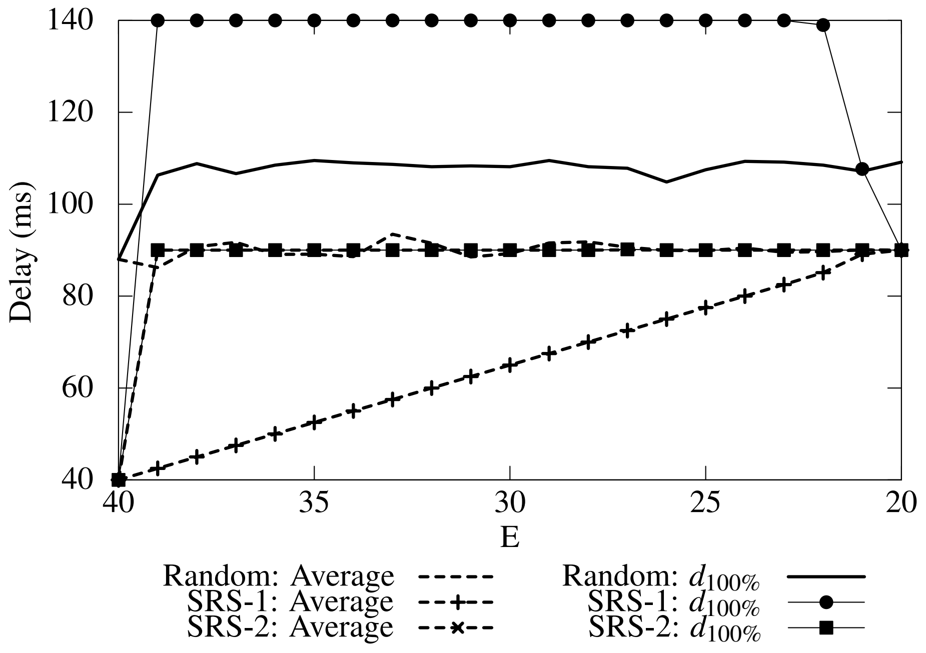


Figure 5.7: Simulation 5.2 results.

ilar to Simulation 5.1. We assumed $20 \le E \le 40$ interacting players are located in the East, and the remaining $40 - E$ players were located in West; the referees are selected from any of the three regions.

As shown in Figure 5.7, when 100% fairness is guaranteed, SRS-2 is significantly better than SRS-1. In contrast, when fairness is not guaranteed, the average delay for SRS-1 is significantly better than SRS-2. Note that the average and $d_{100\%}$ delays are almost identical for SRS-2. Provided P is distributed across many centres in the network, and not confined to a small number of locations as in Simulation 5.1, we believe the results will be comparable to Figure 5.7.

Simulation 5.2 indicates that SRS-1 succeeds in its emphasis on responsiveness and SRS-2 succeeds in its emphasis on fairness. Consequently, a developer can choose either algorithm depending on which criterion is more important.

5.5.3 Simulation 5.3

The inputs to Simulations 1 and 2 do not model the complexity of the Internet or player behaviour. Simulation 5.3 addresses these shortcomings by using the realistic inputs from Section 3.4.3.1. Sets for Z_x^P and P were extracted from the *mshmro.com* March 2007 server log. A set Z_x^P is populated with all participating players (a player is participating if he kills an opponent or is killed) in a randomly selected 10 minute interval. One hundred Z_x^P sets were generated for the simulation. To capture the diurnal behaviour of players, we divide each day of the log into six different 4-hour blocks as in [56] - 12am-4am, 4am-8am, 8am-12pm,12pm-4pm, 4pm-8pm, 8pm-12am. For each 4 hour time period, all 31 blocks were merged to produce a month block (6 month blocks in total). We assume a month block

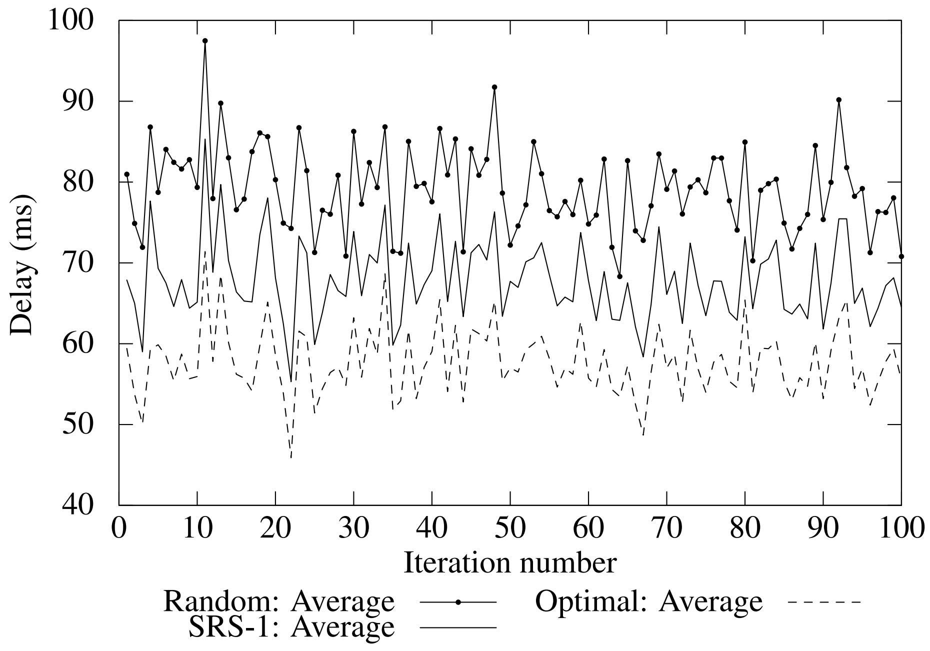


Figure 5.8: Simulation 5.3 average delay.

represents P, the set of players in the MMOG at a given time of day. Finally, we generated 10D [155] network coordinates for all players in the realistic topology by using Vivaldi for 2×10^6 rounds.

The average delays between each peer in Z_x^P and every candidate referee in Z_x^{CR} for SRS-1, random selection, and optimal selection are shown in Figure 5.8. SRS-1 achieves excellent results; they are only $10.12ms$ (18%) slower than optimal on average, whereas random selection is $21.51ms$ (37%) slower. Due to the low average delay of peers in the data set we anticipate even better results in practise. The maximum delays ($d_{100\%}$) between a peer in Z_x^P and a candidate referee in Z_x^{CR} for SRS-2, random selection, and optimal selection are shown in Figure 5.9. While SRS-2 does improve upon random selection, the benefit is small, with SRS-2 and random selection being on average $228.12ms$ (246%) and $284.94ms$ (307%) slower than optimal. Vivaldi produces accurate delay estimations for the majority of nodes; however, it can produce wildly inaccurate results for a small percentage of nodes [91]. We believe this is the reason for the poor results in Figure 5.9, as a single node in either Z_x^P or Z_x^R can increase the maximum delay.

5.6 Summary

Distributing referees to peers has the potential to maximise scalability, responsiveness, and fairness, while minimising the required infrastructure. However, this increases the opportunity for cheating as the publisher loses control of the authoritative game state. This chapter has proposed the Distributed Referee Anti-Cheat Scheme (DRACS), and investigated the Referee Selection Problem (RSP). We formally defined the RSP, the solution being critical

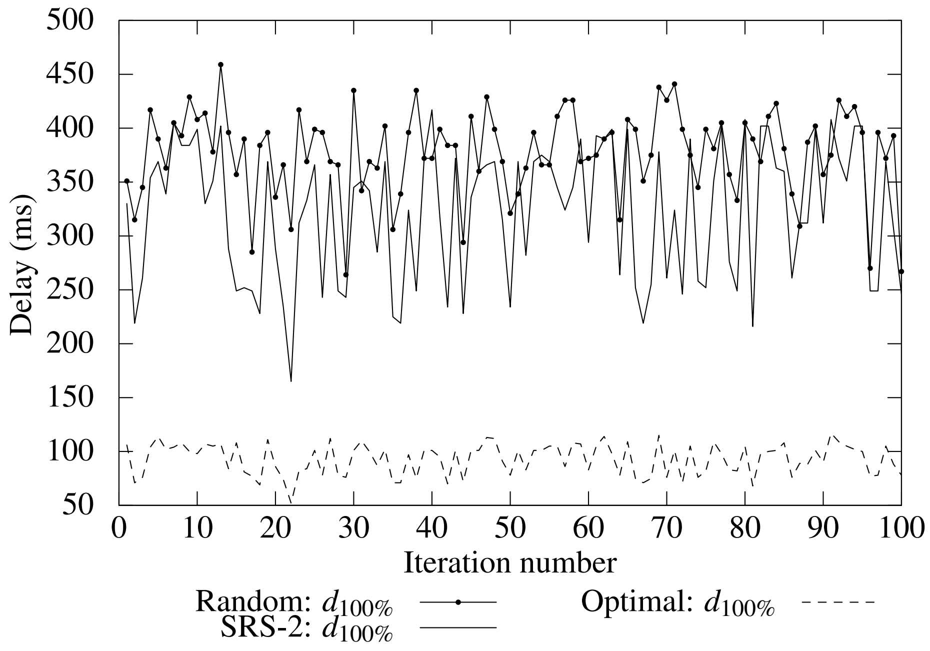


Figure 5.9: Simulation 5.3 maximum delay.

to prevent cheating and improve the performance of P2P network games that use referees to identify cheaters. The RSP raises three criteria in selecting an optimal referee set: security, responsiveness, and fairness. We have argued that the three requirements are conflicting, and therefore proposed two heuristic algorithms, SRS-1 and SRS-2, to solve the RSP.

SRS-1 solves the RSP by selecting referees such that the average peer-to-referee delay is minimised (emphasising responsiveness), while SRS-2 selects referees such that the maximum distance to all players is minimised (emphasising fairness). We have evaluated our algorithms using three simulations and discussed the merits of the solutions. We suggest game developers, first, use Equation (5.1) to calculate the minimum size of the candidate referee selection pool to meet their required level of security. Then, use either SRS-1 or SRS-2 to select referees, depending on the game requirements for responsiveness and fairness.

Two areas currently not addressed by DRACS are reliability and global connectivity. As there is a high probability of referee failure (compared to the servers in C/S, *etc.*), DRACS requires a secure failure recovery mechanism to be practical. Further, as each referee handles only a small portion of the virtual world, DRACS needs a global connectivity mechanism to allow players' avatars to move throughout the world, transitioning between peers. These two problems are left as future work.

Chapter 6

Conclusion

6.1 Summary

This thesis has investigated problems and proposed solutions related to network game cheating, scalability, responsiveness, and fairness. It reviews all known cheats and anti-cheat solutions, and proposes a new cheat classification. The theme of this thesis is the use of trusted referees to increase scalability and responsiveness while preventing cheating. As in traditional sporting events, referees observe the game and only intervene when the rules are not followed. The referee's role is passive, in contrast to the active role played by the server in C/S architectures, and thus it has lower computation and bandwidth requirements than the server in C/S.

In Chapter 3, we proposed the Referee Anti-Cheat Scheme (RACS). Further, we have proposed two centralised round length adjustment algorithms, an optimal *brute force* approach, and a faster *voting* algorithm. For architectures with a trusted third party, these algorithms are preferable to the distributed approach in [64]. We used analytical analysis to show that RACS has lower delay and bandwidth than C/S, NEO [64], and SEA [36]; and cheat prevention/detection equal to that in C/S (exceeding that in NEO and SEA). Further, we used simulation to evaluate RACS and our round length adjustment algorithms. Since there are currently no publicly available traces of MMOG, Chapter 3 also describes how we constructed two classes of inputs, artificial and realistic, used in the simulations throughout this thesis. The artificial inputs are simple and unrealistic, but clearly demonstrate system behaviour. The realistic inputs are complex, and constructed from real-world traces of network games and Internet measurements. Using the artificial and realistic inputs our simulations show that RACS has lower delay and bandwidth than C/S, and that RACS remains effective when packet loss and cheating occur. Finally, we use simulation to show the effectiveness of our round length adjustment algorithms. While RACS has far better scalability than C/S, with only one trusted referee it may still suffer a system bottleneck.

In Chapter 4, we proposed the Mirrored Referee Anti-Cheat Scheme (MRACS) that combines the favourable features of the RACS and Mirrored Server (MS) [41–43] architectures. MRACS has better responsiveness, fairness, and bandwidth scalability than RACS and MS, while removing RACS's single point of failure, addressing cheating in MS, and reducing processing costs in MS. MRACS, unlike RACS, requires the publisher to provision multiple servers connected via a private network, each hosting a mirrored referee. Chapter 4 also discusses the Client-to-Mirror Assignment (CMA) problem for mirrors with limited resources; which is applicable to MRACS, MS, and other applications involving mirrored resources with long term connections (*e.g.*, video streaming). We have proposed the optimal J-SA/L-SA and the faster heuristic J-Greedy/L-Greedy; all algorithms run in polynomial time. Our analytical and simulation results show that MRACS has better scalability and responsiveness than RACS and MS, and thus C/S. Further, we have used simulation to show that J-SA/L-SA and J-Greedy/L-Greedy produce significantly lower client-to-mirror delays than the existing J-Greedy/L-Ignore.

Finally, in Chapter 5, we proposed the Distributed Referee Anti-Cheat Scheme (DRACS) that distributes referees to peers to maximise scalability, responsiveness, and fairness; while minimising the required publisher provisioned infrastructure. Like in references [30, 58, 109, 154], DRACS assumes the virtual world is partitioned into zones. DRACS selects a set of referees, each running on an untrusted peer, to prevent and/or detect cheating among the players in the zone. We have formally defined the Referee Selection Problem (RSP) to securely select referees such that responsiveness and/or fairness are maximised, while preventing cheating. We have proposed two centralised solutions for RSP, SRS-1 and SRS-2, which maximise responsiveness and fairness respectively. We use simulation to show that both algorithms are far superior to random selection, which is equivalent to the distributed approach proposed by Corman *et al.* [37].

6.2 Future Work

While considerable effort has been made to ensure the simulations and analysis are as realistic as possible, real implementations and evaluations of RACS, MRACS, and DRACS on games of different genres would strengthen this work. Note, an early version of the RACS protocol has been implemented and evaluated by Palmer [107].

RACS provides cheat prevention equal to that in C/S; however, the cost of digitally signing every message consumes considerable bandwidth and processing power. The costly digital signatures are used to prevent the inconsistency cheat, which is rare in current commercial games. One may investigate replacing the digital signatures with encryption for messages sent to and from the referees (MPR and MRP), reducing bandwidth and processing costs while maintaining the protocol's cheat-free features. Further, in many games it may be sufficient to detect the occurrence of the inconsistency cheat, without determining which player is cheating. For this case, we believe one can implement a simplified RACS protocol without using digital signatures.

While several protocols have been proposed which divide time into rounds (*e.g.*, AS, NEO, RACS, MRACS, *etc.*), very little work has been done to evaluate how round length influences player performance and enjoyment in commercial games. It is also worth investigating improving the performances of RACS, MRACS, and DRACS by allowing different round lengths for different players, depending on their network capacity.

In Chapter 4 we did not consider the mirror placement problem [81, 117, 129] as it is heavily influenced by business considerations [100]. However, this is an important problem for minimising client-to-mirror delay. Thus, investigating strategies to aid publishers and developers in locating mirrors is a potential area of future research.

Our proposed DRACS architecture in Chapter 5 does not address mechanisms for recovering from failures, partitioning the virtual world, and maintaining global connectivity. We leave these issues for future work.

Appendix A

L-SA Optimality Proof

The following proof was developed by Jerry Trahan[1].

The L-SA algorithm updates the Client-to-Mirror Assignment (CMA) when a client leaves. L-SA is optimal, that is, if the CMA with n clients, CMA_n, had minimum total client-to-mirror delay (minimum $delay(CMA_n)$) among all possible assignments before client P_i leaves, then the CMA with $n-1$ clients, CMA_{n-1}, produced by L-SA after P_i leaves has minimum $delay(CMA_{n-1})$. The optimality proof has the following structure. First, function *construct_labels_leave*() is optimal in that it constructs, for each occupied (*i.e.*, non-empty) mirror M_O, the least cost chain of player reassignments from M_O to M_X, where M_X is the mirror from which P_i left, if a negative cost chain exists. Second, let CMA_n^- denote CMA_n with P_i removed. For any alternative assignment $CMA_{n-1}^* \neq CMA_{n-1}$, we can construct a transformation graph G showing the changes in client assignments between CMA_n^- and CMA_{n-1}^*. We will decompose G into cycles and paths, where each path corresponds to a sequence of client moves ending at a mirror that was not full under CMA_n^-. We will show that each cycle and each path (except possibly a path ending at M_X) must either not change the overall total client-to-mirror delay or increase the delay.

Notation: This proof uses the same notation as in Chapter 4, but is repeated here for convenience. Player P_i is the leaving player, and, CMA_n assigned P_i to M_X. The transfer heaps array TH[1...m, 1..m] is an array of min heaps. Min heap TH[R, S] contains tuples (*cost*, *player*), where *cost* is the delay offset of transferring player from M_R to M_S (*i.e.,* $d_{j,S} - d_{j,R}$ for player P_j). Array element Label[Y] is a pair (*cost*, *mirror*) representing a chain starting from M_Y and ending at M_X: *mirror* is the next mirror in the chain after M_Y; for a pair of mirrors M_R and M_S in sequence along this chain, the minimum cost tuple of TH[R, S] identifies the player P_j to move from M_R to M_S and identifies the change in delay due to this move; and *cost* is the additional delay (called chain cost) of reassigning players along this chain.

Function *construct_labels_leave*() essentially runs the Bellman-Ford single-source short-

[1]Jerry L. Trahan, email to the author, June 29th, 2010.

est paths (SSSP) algorithm. In the SSSP graph, each vertex corresponds to a mirror, and the weight of edge (u, v) is the minimum cost of transferring a player from M_u to M_v (*i.e.*, TH[u, v].*cost*). Rather than finding the shortest path from a source s to each vertex in a graph, *construct_labels_leave*() reverses direction to find the least cost path with negative cost from each mirror to mirror M_X. Also, it omits some edges that have no bearing on the result.

Lemma 1. *Given an optimal CMA_n and a player leaving M_X, construct_labels_leave() finds for each occupied mirror M_O, the negative cost chain to M_X with least cost, if a negative cost chain exists.*

Proof. We argue the optimality of *construct_labels_leave*() with the optimality of the Bellman-Ford SSSP algorithm. Function *construct_labels_leave*() runs Bellman-Ford on a graph with a vertex for each mirror and the weight of edge from vertex u to vertex v is equal to the minimum change in delay from transferring a player from M_u to M_v. Rather than dealing with the complete graph, it limits its search by omitting edges that cannot be part of any negative cost path. Consequently, the algorithm will find shortest paths to vertex X (least cost chains to M_X) for the graph it processes, so the rest of the proof establishes that omitting some edges does not change the result if a negative-cost chain exists.

The initialisation process sets Label[X] to (0, -), and sets Label[Y] to (∞, -) for all other mirrors $M_Y \in M$. The "graph" on which *construct_labels_leave*() finds shortest paths has a vertex for each mirror and edges from vertices corresponding to $M_O \in M$, where M_O is occupied (not empty) and $O \neq X$, to vertices corresponding to $M_Y \in M$, where $Y \neq O$ and (M_Y is full or $Y = X$). This "graph" omits the following edges:

(i) edges originating from empty (unoccupied) mirrors,

(ii) edges originating from M_X (*i.e.*, $M_O = M_X$),

(iii) self-loops (*i.e.,* $M_Y = M_O$), and

(iv) edges to non-full mirrors other than M_X.

The omitted edges cannot be on any negative cost path (chain) for the following reasons:

(i) If a mirror is empty, then it has no players to transfer to other mirrors. Also, no reason exists to consider moving a player through an empty mirror M_E, since the change in delay in moving a player from M_O to M_E to M_Y is the same as moving the player directly from M_O to M_Y.

(ii) If $M_O = M_X$, then this corresponds to a chain starting from M_X, forming a cycle. Because CMA_n was optimal, no negative cost cycle can exist.

(iii) A self loop does not change the assignment or total client-to-mirror delay.

(iv) If a chain to a non-full mirror (other than M_X) has negative cost, then updating CMA_n by making the transfers along that chain would result in a lower cost assignment, contradicting the optimality of CMA_n. □

Theorem 1. *Algorithm L-SA is optimal.*

Proof. Let CMA_n denote the client-to-mirror assignment before player P_i leaves. Assume that CMA_n was optimal, that is, it had minimum total client-to-mirror delay among all possible client-to-mirror assignments. Let CMA_n^- denote CMA_n with P_i removed. Let CMA_{n-1} denote the client-to-mirror assignment produced by L-SA after P_i leaves, and CMA_{n-1}^* be any alternative assignment. We will prove that CMA_{n-1} has minimum total client-to-mirror delay among all possible client-mirror assignments (*i.e.*, $delay(CMA_{n-1}^*) \geq delay(CMA_{n-1})$ for all possible CMA_{n-1}^*). For any assignment $CMA_{n-1}^* \neq CMA_{n-1}$ there are five possible cases:

1. M_X was not full under CMA_n, and M_X is not full under CMA_{n-1}^*;
2. M_X was not full under CMA_n, and M_X is full under CMA_{n-1}^*;
3. M_X was full under CMA_n, M_X is not full under CMA_{n-1}^*, and $Label[Z].cost \geq 0$;
4. M_X was full under CMA_n, M_X is full under CMA_{n-1}^*, and $Label[Z].cost \geq 0$; and
5. M_X was full under CMA_n, and $Label[Z].cost < 0$.

For case 1, after removing P_i from M_X, if M_X's available capacity is greater than 1, that is, M_X was not full before P_i left, then L-SA simply removes P_i and returns CMA_n^- as CMA_{n-1}. If some other assignment CMA_{n-1}^* had smaller delay than CMA_{n-1}, and non-full M_X, then $delay(CMA_{n-1}^*) + d_{i,X}$ would be smaller than $delay(CMA_n)$, contradicting the optimality of CMA_n. Similarly for case 3, if after removing P_i from M_X, M_X's available capacity is 1 but Label[Z].$cost \geq 0$ (Z is the label with minimum cost), then L-SA again removes P_i and returns CMA_n^- as CMA_{n-1}. By the same argument, CMA_{n-1} is optimal considering all assignments CMA_{n-1}^* with non-full M_X. We will examine cases 2 and 4 at the end of the proof.

The next portion of the proof deals with the case in which M_X was full before P_i left and Label[Z].$cost$ < 0 (case 5). Consider some assignment CMA_{n-1}^* for the players after P_i has left. Construct a transformation graph G describing the player movements to transform CMA_n^- to CMA_{n-1}^*. We will prove that the weight of edges in G is greater than or equal to Label[Z].$cost$, so $delay(CMA_{n-1}^*) \geq delay(CMA_{n-1})$. Graph G is a directed, weighted multigraph with a vertex u for each mirror M_u. If a player P_j connects to M_u under $CMA_n^-$and to M_v under CMA_{n-1}^*, then G contains an edge (u, v) with weight $d_{j,v} - d_{j,u}$. Observe that if multiple players connect to M_u under CMA_n^- and to M_v under CMA_{n-1}^*, then G contains multiple edges from u to v, each with weight corresponding to the delay difference. For a full mirror M_F in CMA_n^-, $indegree(F) \leq outdegree(F)$; otherwise, M_F

would be overloaded in CMA^*_{n-1}. For a non-full mirror M_S in CMA^-_n, $indegree(S)$ can be less than, equal to, or greater than $outdegree(S)$.

We now remove cycles from G and decompose the remaining graph into paths that end at vertices corresponding to non-full mirrors in CMA^-_n. Find a simple cycle c in G, remove the edges in c, and repeat until no more cycles remain. Let H denote the directed acyclic graph remaining after removing the cycles. Decompose H into set K of paths as follows. Topologically sort H, then repeat the following until no more edges remain. Start from the lowest indexed vertex with an outgoing edge. Select an outgoing edge from this vertex to extend the path (choose an arbitrary edge if more than one outgoing edge exists), and repeat from the new endpoint until reaching a vertex with no outgoing edges. Add this path to K and remove its edges from H.

For a simple cycle c removed from G, suppose that $cost(c) < 0$. This cycle describes a reassignment of one player to each mirror and one player from each mirror on the cycle, so it overloads no mirror. If we make the same sequence of reassignments to CMA_n, then the resulting assignment has a delay less than $delay(CMA_n)$, contradicting the optimality of CMA_n, so $cost(c) \geq 0$.

For each path $k_e \in K$, we will prove three properties:

1. k_e ends at a vertex corresponding to a non-full mirror in CMA^-_n;
2. if k_e ends at vertex Y such that $M_Y \neq M_X$, then $cost(k_e) \geq 0$ (*i.e.*, any path not ending at M_X cannot reduce the total delay); and
3. if k_e ends at vertex X corresponding to M_X, then $cost(k_e) \geq$ Label[Z].$cost$ in L-SA (*i.e.*, all paths ending at M_X have equal or greater delay than the transfer chain from M_Z to M_X in L-SA.

For property 1, let L denote the set of vertices in H with in-degree less than or equal to out-degree. Set L contains all vertices corresponding to full mirrors in CMA^-_n, and it may contain vertices corresponding to non-full mirrors. Consider the first path $k_0 =< v_0, v_1, ..., v_q >$ at the time of its construction. By definition, v_q has at least one incoming edge and no outgoing edge, so $v_q \in V - L$ and corresponds to a non-full mirror. To show that the conditions on vertices in L hold as the path construction proceeds, we want to show that if $v_t \in L$ before removing the edges in k_0, then $indegree(v_t) \leq outdegree(v_t)$ still holds after removing the edges in k_0. If $v_t \notin k_0$, then its degrees do not change. If $v_t = v_0$, then its in-degree was 0 before constructing k_0, so $indegree(v_t) \leq outdegree(v_t)$ still holds after removing the edges in k_0. If $v_t \in \{v_1, v_2, ..., v_{q-1}\}$, then removing edges in k_0 decreases both in-degree and out-degree by one. Finally, v_t cannot be v_q because $v_q \in V - L$. Because the conditions on in-degree and out-degree of vertices in L hold before creating k_0 and after removing the edges in k_0, then they hold through the construction of all other paths, so, like $v_q \in k_0$, each path in K ends at a vertex corresponding to a non-full mirror.

For property 2, suppose $k_e =< v_0, v_1, ..., v_q >$ ends at vertex Y such that $M_Y \neq M_X$. Assume that $cost(k_e) < 0$. Since $M_Y \neq M_X$ and property 1, M_Y was a non-full mirror in CMA_n. Therefore, the sequence of player transfers described by k_e could be made to CMA_n, ending with an overall delay smaller than that of CMA_n. This contradicts the optimality of CMA_n, so $cost(k_e) \geq 0$.

For property 3, suppose k_e ends at M_X. The chain of player reassignments made by L-SA has cost Label[Z].*cost* < 0; by Lemma 1, this chain has the least cost among all chains of reassignments ending at M_X. Consequently, $cost(k_e) \geq$ Label[Z].*cost*.

This completes the proof that when M_X was full before P_i left and Label[Z].*cost* < 0 (case 5).

We return now to case 2, where M_X was not full before P_i left and consider alternative assignments CMA^*_{n-1} with full M_X. The transformation graph argument given above applies up to the three properties. Property 1 and its proof hold in this case. Replace Properties 2 and 3 with Property 2a: $cost(k_e) \geq 0$. Suppose $k_e =< v_0, v_1, ..., v_q >$ ends at vertex Y. Assume that $cost(k_e) < 0$. If $M_Y \neq M_X$, then property 1, M_Y was a non-full mirror in CMA_n. If $M_Y = M_X$, then for this case M_Y was a non-full mirror in CMA_n. Therefore, the sequence of player transfers described by k_e could be made to CMA_n, ending with an overall delay smaller than that of CMA_n. This contradicts the optimality of CMA_n, so $cost(k_e) \geq 0$, and this completes the proof for this case.

Finally, we return to case 4, where M_X was full before P_i left, M_X is full under CMA^*_{n-1}, and Label[Z].*cost* ≥ 0. The transformation graph argument given above applies up to the three properties. Property 1 and its proof hold again in this case. Replace Properties 2 and 3 with Property 2a: $cost(k_e) \geq 0$. The proof given above for Property 2 holds here where $M_Y \neq M_X$. If $M_Y = M_X$, then by Lemma 1 since Label[Z].*cost* ≥ 0, no negative cost chain to M_X exists, so $cost(k_e) \geq 0$, and this completes the proof for this case.

Collecting these facts about cycles and paths in the decomposition, $delay(CMA^*_{n-1})$ $\geq delay(CMA_{n-1})$ for any assignment CMA^*_{n-1}. $\square$

Appendix B

Copyright Permissions

This thesis is based upon several works that have been published over the course of the author's PhD. Except for two publications, the copyright agreements for my papers allow the authors to re-use their material in derivative works; thus, copyright permission is not required. The following permissions were obtained to allow the re-use of material from these two papers.

• Webb, S. D., W. Lau, and S. Soh (2006). NGS: An application layer network game simulator. In *Proc. Interactive Entertainment (IE)*, pp. 15-22.

From: K.Wong@murdoch.edu.au
To: steven.webb@postgrad.curtin.edu.au
Date: 4 September 2009
Subject: RE: Copyright permission for CGIE06
Hi Steve,

Just realised that the copyright belongs to me (as one of the editors of the proceedings) at Murdoch university. In this case, it is ok for you to include in your thesis as long as you also make reference to the proceedings. The electronic version of the proceedings are at the following:

http://portal.acm.org/citation.cfm?id=1234341

http://portal.acm.org/citation.cfm?id=1231894

Cheers,

Kevin

• Webb, S. D., S. Soh, and J. L. Trahan (2009). Secure referee selection for fair and responsive peer-to-peer gaming. *SIMULATION: Transactions of The Society for Modelling and Simulation International 85*(9), 608-618.

From: Marta.Granatowska@sagepub.co.uk
To: steven.webb@postgrad.curtin.edu.au
Date: 3 July 2009
Subject: RE: Permission to reuse work

Dear Steven,
Thank you for your e-mail. Please consider this e-mail to be written permission for the below request.
Kind regards,
Marta Granatowska
Rights Executive
SAGE Publications Ltd

From: steven.webb@postgrad.curtin.edu.au
To: permissions@sagepub.com
Date: 2 July 2009
Subject: Permission to reuse work

Sage permissions,
I am currently writing my PhD thesis and would like to use large extracts copied verbatim from an article I had published with SAGE: Webb, S. D., S. Soh, and J. L. Trahan (2009). Secure referee selection for fair and responsive peer-to-peer gaming. *SIMULATION: Transactions of The Society for Modelling and Simulation International 85*(9), 608-618. This article is currently available though OnlineFirst.

It is a requirement of my degree that my thesis is made publicly available via the Australian Digital Thesis Collection (http://adt.caul.edu.au). The material will be provided strictly for educational purposes and on a non-commercial basis. Full acknowledgement of the ownership of the copyright and the original source of the material will be provided with the material. I would be willing to use a specific form of acknowledgement that you may require and to communicate any conditions relating to its use.
Kind regards,
Steven Daniel Webb.

Bibliography

[1] Diablo III. http://www.diablo3.com/.

[2] Fallout 3. http://fallout.bethsoft.com/.

[3] WoW Realm Status. web page. *http://www.wowrealmstatus.net/.*

[4] Abadi, M. and R. Needham (1996). Prudent engineering practice for cryptographic protocols. *IEEE Trans. Software Engineering 22*(1), 6–15.

[5] Abdelkhalek, A., A. Bilas, and A. Moshovos (2003). Behavior and performance of interactive multi-player game servers. *Cluster Computing 6*, 355–366.

[6] Aggarwal, S., H. Banavar, S. Mukherjee, and S. Rangarajan (2005). Fairness in dead-reckoning based distributed multi-player games. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 1–10.

[7] AhnLab. Hackshield. web page. *http://www.hackshields.com/.*

[8] Alexander, T. (Ed.) (2005). *Massively Multiplayer Game Development 2*. Hingham, Massachusetts: Charles River Media, Inc.

[9] Arfken, G. (1985). *Mathematical Methods for Physicists* (3 ed.). Academic Press.

[10] Armitage, G. (2003). An experimental estimation of latency sensitivity in multiplayer Quake 3. In *Proc. International Conference on Networks (ICON)*, pp. 137–141.

[11] Armitage, G. (2008a). Client-side adaptive search optimisation for online game server discovery. In *Proc. IFIP/TC6 NETWORKING*, pp. 494–505.

[12] Armitage, G. (2008b). Discovering first person shooter game servers online: techniques and challenges. *International Journal of Advanced Media and Communication 2*(4), 402–414.

[13] Baughman, N. E., M. Liberatore, and B. N. Levine (2006). Cheat-proof playout for centralized and peer-to-peer gaming. *IEEE/ACM Trans. Networking 15*(1), 1–13.

[14] Beigbeder, T., R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool (2004). The effects of loss and latency on user performance in unreal tournament 2003. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 144–151.

[15] Bernier, Y. W. (2001). Latency compensating methods in client/server in-game protocol design and optimization. In *Proc. Game Developers Conference (GDC).*

[16] Bettner, P. and M. Terrano (2001). 1500 archers on a 28.8: Network programming in Age of Empires and beyond. In *Proc. Game Developers Conference (GDC).*

[17] Bharambe, A., J. R. Douceur, J. R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang (2008). Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In *Proc. SIGCOMM*, pp. 389–400.

[18] Blizzard (2002, Aug). Map hack. web page. *http://www.blizzard.com/support/?id=nNews054p.*

[19] Blizzard Entertainment (2006). World of Warcraft. web page. *http://www.worldofwarcraft.com/* Last accessed on the 27th June 2006.

[20] Blizzard Entertainment (2008a, Nov). World of warcraft subscriber base reaches 11.5 million worldwide. Press Release. *http://us.blizzard.com/en-us/company/press/pressreleases.html?081121.*

[21] Blizzard Entertainment (2008b, Jan). World of warcraft surpasses 9 million subscribers worldwide. Press Release. *http://www.blizzard.com/ press/070724.shtml.*

[22] Boorstyn, R. R. and H. Frank (1977). Large-scale network topological optimization. *IEEE Trans. Communications 25*(1), 29–47.

[23] Brack, J. A. and F. Pearce (2009, September). GDC Austin: An inside look at the universe of Warcraft. Conference talk. *http://www.gamasutra.com/php-bin/news_index.php?story=25307.*

[24] Brandt, D. (2005). Networking and scalability in EVE Online. Slide Show. *http://www.research.ibm.com/netgames2005/papers/brandt.pdf* Last accessed on the 7th March 2006.

[25] Brun, J., F. Safaei, and P. Boustead (2006). Fairness and playability in online multiplayer games. In *Proc. Consumer Communications and Networking Conference (CCNC)*, pp. 1199–1203.

[26] Caldwell, P. (2006, July). Blizzarrd bans 59,000 WOW accounts. web page. *http://au.gamespot.com/news/6154708.html.*

[27] Cecin, F., R. Real, R. de Oliveira Jannone, C. R. Geyer, M. Martins, and J. V. Barbosa (2004). FreeMMG: A scalable and cheat-resistant distribution model for Internet games. In *Proc. Distributed Simulation and Real Time Applications (DS-RT)*, pp. 83–90.

[28] Chambers, C., W. C. Feng, S. Sahu, and D. Saha (2005). Measurement-based characterization of a collection of on-line games. In *Proc. Internet Measurement Conference (IMC)*, pp. 1–14.

[29] Chen, B. D. and M. Maheswaran (2004). A cheat controlled protocol for centralized online multiplayer games. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 139–143.

[30] Chen, J., B. Wu, M. Delap, B. Knutsson, H. Lu, and C. Amza (2005). Locality aware dynamic load management for massively multiplayer games. In *Proc. Principles and Practice of Parallel Programming (PPoPP)*, pp. 289–300.

[31] Chen, K.-T., P. Huang, and C.-L. Lei (2006). Game traffic analysis: an MMORPG perspective. *Computer Networks 50*(16), 3002–3023.

[32] Chen, K.-T., P. Huang, and C.-L. Lei (2009). Effect of network quality on player departure behavior in online games. *IEEE Trans. Parallel and Distributed Systems (TPDS) 20*(5), 593–606.

[33] Chen, K. T. and C. L. Lei (2006). Network game design: Hints and implications of player interaction. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 1–9.

[34] Claypool, M. (2008). Network characteristics for server selection in online games. In *Proc. Multimedia Computing and Networking (MMCN)*, pp. 6818–07.

[35] Cohen, B. (2003). Incentives build robustness in BitTorrent. In *Proc. Economics of Peer-to-Peer Systems P2PECON*.

[36] Corman, A. B., S. Douglas, P. Schachte, and V. Teague (2006). A Secure Event Agreement (SEA) protocol for peer-to-peer games. In *Proc. ARES*, pp. 34–41.

[37] Corman, A. B., P. Schachte, and V. Teague (2007). A secure group agreement (sga) protocol for peer-to-peer applications. In *Proc. Advanced Information Networking and Applications Workshops (AINAW)*, pp. 24–29.

[38] Cottrell, R. L. and S. Khan (2007). CFA SCIC network monitoring report. web page. *http://www.slac.stanford.edu/xorg/icfa/icfa-net-paper-jan07/*.

[39] Counter Hack (2007, Mar). Halflife. web page. *http://wiki.counter-hack.net/halflife*.

[40] Cronin, E., B. Filstrup, and S. Jamin (2003). Cheat-proofing dead reckoned multiplayer games. In *Proc. Applicatoin and Development of Computer Games (ADCOG)*.

[41] Cronin, E., B. Filstrup, and A. Kurc (2001, May). A distributed multiplayer game server system. Project Report. *http://warriors.eecs.umich.edu/games/papers/quakefinal.pdf*.

[42] Cronin, E., B. Filstrup, A. R. Kurc, and S. Jamin (2002). An efficient synchronization mechanism for mirrored game architectures. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 67–73.

[43] Cronin, E., A. R. Kurc, B. Filstrup, and S. Jamin (2004). An efficient synchronization mechanism for mirrored game architectures. *Multimedia Tools and Applications 23*(1), 7–30.

[44] Dabek, F., R. Cox, F. Kaashoek, and R. Morris (2004). Vivaldi: a decentralized network coordinate system. In *Proc. SIGCOMM*, pp. 15–26.

[45] Davis, S. (2007, Feb). Next-gen hacking / last-gen weaknesses - part 1 - gears of war for the xbox 360. web page. *http://playnoevil.com/serendipity/index.php?/archives/1123-Next-Gen-Hacking-Last-Gen-Weaknesses-Part-1-Gears-of-War-for-the-Xbox-360.html*.

[46] Davis, S. (2009). *Protecting games: a security handbook for game developers and publishers*. Charles River Media.

[47] DeLap, M., B. Knutsson, H. Lu, O. Sokolsky, U. Sammapun, I. Lee, and C. Tsarouchis (2004). Is runtime verification applicable to cheat detection? In *Proc. Network and Systems Support for Games (NetGames)*, pp. 134–138.

[48] Dick, M., O. Wellnitz, and L. Wolf (2005). Analysis of factors affecting players' performance and perception in multiplayer games. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 1–7.

[49] Dyck, J. (2006, January). A survey of application-layer networking techniques for real-time distributed groupware. PhD Comprehensive Exam Survey.

[50] Even Balance (2002). PunkBuster online countermeasures. web page. *http://www.evenbalance.com/* Last accessed on the 26th September 2006.

[51] Falkner, J., M. Piatek, J. P. John, A. Krishnamurthy, and T. Anderson (2007). Profiling a million user dht. In *Proc. Internet Measurement Conference (IMC)*, pp. 129–134.

[52] Fan, L., H. Taylor, and P. Trinder (2007). Mediator: a design framework for P2P MMOGs. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 43–48.

[53] Fei, Z., S. Bhattacharjee, E. Zegura, and M. Ammar (1998). A novel server selection technique for improving the response time of a replicated service. In *Proc. INFOCOM*, pp. 783–791.

[54] Feng, W. C., F. Chang, W. C. Feng, and J. Walpole (2002). Provisioning on-line games: a traffic analysis of a busy counter-strike server. In *Proc. ACM SIGCOMM Workshop on Internet measurment (IMW)*, pp. 151–156.

[55] Feng, W. C., F. Chang, W. C. Feng, and J. Walpole (2005). A traffic characterization of popular on-line games. *ACM/IEEE Trans. on Networking 13*(3), 488–500.

[56] Feng, W. C. and W. C. Feng (2003). On the geographic distribution of on-line game servers and players. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 173–179.

[57] Feng, W. C., E. Kaiser, and T. Schluessler (2008). Stealth measurements for cheat detection in on-line games. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 15–20.

[58] Fernandes, S., C. Kamienski, D. Sadok, J. Moreira, and R. Antonello (2007). Traffic analysis beyond this world: the case of second life. In *Proc. Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pp. 43–48.

[59] Ferretii, S., C. Palazzi, M. Roccetti, G. Pau, and M. Gerla (2006). Buscar el levante port el poniente: in search of fairness through interactivity in massively multiplayer online games. In *Proc. Consumer Communications and Networking Conference (CCNC)*, pp. 1183–1187.

[60] Francis, P., S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang (2001). Idmaps: a global internet host distance estimation service. *IEEE/ACM Trans. Networking 9*(5), 525–540.

[61] Fritsch, T., H. Ritter, and J. Schiller (2005). The effect of latency and network limitations on mmorpgs: a field study of everquest2. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 1–9.

[62] GauthierDickey, C., V. Lo, and D. Zappala (2005). Using n-trees for scalable event ordering in peer-to-peer games. In *Proc. Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pp. 87–92.

[63] GauthierDickey, C., D. Zappala, and V. Lo (2004). Low latency and cheat-proof event ordering for distributed games. Technical Report CIS-TR-2004-2, University of Oregon.

[64] GauthierDickey, C., D. Zappala, V. Lo, and J. Marr (2004). Low latency and cheat-proof event ordering for peer-to-peer games. In *Proc. Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pp. 134–139.

[65] Gautier, L., C. Diot, and J. Kurose (1999). End-to-end transmission control mechanisms for multiparty interactive applications on the Internet. In *Proc. INFOCOM*, pp. 1470–1479.

[66] Gianvecchio, S., Z. Wu, M. Xie, and H. Wang (2009). Battle of botcraft: fighting bots in online games with human observational proofs. In *Proc. Computer and Communications Security (CCS)*, pp. 256–268.

[67] Goodman, J. and C. Verbrugge (2008). A peer auditing scheme for cheat elimination in mmogs. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 9–14.

[68] Gornall, S. (2008). Ip address lookup – community geotarget IP project. web page. *http://www.hostip.info*.

[69] Guo, K., S. Mukherjee, S. Rangarajan, and S. Paul (2003). A fair message exchange framework for distributed multi-player games. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 29–41.

[70] Hsiao, T.-Y. and S.-M. Yuan (2005). Practical middleware for massively multiplayer online games. *IEEE Internet Computing 9*(5), 47–54.

[71] Hu, S., J. Chen, and T. Chen (2006). VON: A scalable peer-to-peer network for virtual environments. *IEEE Network 20*(4), 22–31.

[72] Hulu. http://www.hulu.com/.

[73] id Software (1998, Dec). Quake World. web page. *http://www.quakeworld.net/* Last accessed on the 27th June 2006.

[74] Iimura, T., H. Hazeyama, and Y. Kadobayashi (2004). Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 116–120.

[75] Johnson, D. B. and D. A. Maltz (1996). Dynamic source routing in ad hoc wireless networks. *Mobile Computing 353*, 153–181.

[76] Kabus, P. and A. Buchmann (2007). Design of a cheat-resistant P2P online gaming system. In *Proc. Digital Interactive Media in Entertainment and Arts (DIMEA)*, pp. 113–120.

[77] Kabus, P., W. W. Terpstra, M. Cilia, and A. Buchmann (2005). Addressing cheating in distributed MMOGs. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 1–6.

[78] Kangasharju, J., J. Robers, and K. W. Ross (2002). Object replication strategies in content distribution networks. *Computer Communications 25*(4), 376–383.

[79] Kawahara, Y., T. Aoyama, and H. Morikawa (2004). A peer-to-peer message exchange scheme for large-scale networked virtual environments. *Telecommunication Systems 25*(3), 353–370.

[80] Keller, J. and G. Simon (2003). Solipsis: A massively multi-participant virtual world. In *Parallel and Distributed Processing Techniques and Applications*, pp. 262–268.

[81] Kershenbaum, A. (1993). *Telecommunication Network Design Algorithms*. McGraw-Hill.

[82] Khuri, S. and T. Chiu (1997). Heuristic algorithms for the terminal assignment problem. In *Proc. Symposium on Applied computing (SAC)*, pp. 247–251.

[83] Kim, J., J. Choi, D. Chang, T. Kwon, Y. Choi, and E. Yuk (2005). Traffic characteristics of a massively multi-player online role playing game. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 1–8.

[84] Knutsson, B., H. Lu, W. Xu, and B. Hopkins (2004). Peer-to-peer support for massively multiplayer games. In *Proc. INFOCOM*, Volume 1, pp. 7–11.

[85] Kushner, D. (2005). Engineering Everquest: Online gaming demands heavyweight data centers. *IEEE Spectrum 42*(7), 34–39.

[86] Lakshminarayanan, K. and V. N. Padmanabhan (2003). Some findings on the network performance of broadband hosts. In *Proc. Internet Measurement Conference (IMC)*, pp. 45–50.

[87] Lee, J. (2005, May). Wage slaves. web page. *http://www.1up.com/do/feature?cId=3141815*.

[88] Lee, K. W., B. J. Ko, and S. Calo (2005). Adaptive server selection for large scale interactive online games. *Computer Networks 49*(1), 84–102.

[89] Li, K., S. Ding, D. McCreary, and S. Webb (2004). Analysis of state exposure control to prevent cheating in online games. In *Proc. Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pp. 140–145.

[90] Lua, E. K., J. Crowcroft, M. Pias, R. Sharma, and S. Lim (2005). A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, 72–93.

[91] Lua, E. K., T. Griffin, M. Pias, H. Zheng, and J. Crowcroft (2005). On the accuracy of embeddings for Internet coordinate systems. In *Proc. Internet Measurement Conference (IMC)*, pp. 125–138.

[92] Lui, J. and M. Chan (2002). An efficient partitioning algorithm for distributed virtual environment systems. *IEEE Trans. on Parallel and Distributed Systems 13*(3), 193–211.

[93] Matthews, W. and L. Cottrell (2000). The PingER project: active Internet performance monitoring for the HENP community. *IEEE Communications Magazine 38*(5), 130–136.

[94] McCoy, A., T. Ward, S. Mcloone, and D. Delaney (2007). Multistep-ahead neural-network predictors for network traffic reduction in distributed interactive applications. *Trans. on Modeling and Computer Simulation (TOMACS) 17*(4), 16.

[95] MDY Industries (2007). Glider. web page. *http://www.wowglider.com/*.

[96] Miller, J. L. and J. Crowcroft (2009). Probabilistic event resolution with the pairwise random protocol. In *Proc. Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pp. 67–72.

[97] Mills, D. (1992, March). Network time protocol. RFC 1305.

[98] Mönch, C., G. Grimen, and R. Midtstraum (2006). Protecting online games against cheating. In *Proc. Network and Systems Support for Games (NetGames)*. Article number 20.

[99] Montresor, A. (2004). A robust protocol for building superpeer overlay topologies. In *Proc. Peer-to-Peer Computing (P2P)*, pp. 202–209.

[100] Mulligan, J. and B. Patrovsky (2003, February). *Developing Online Games: An Insider's Guide*. New Riders Publishing.

[101] Neumann, C., N. Prigent, M. Varvello, and K. Suh (2007). Challenges in peer-to-peer gaming. *Proc. SIGCOMM 37*(1), 79–82.

[102] Ng, T. S. E. and H. Zhang (2002). Predicting internet network distance with coordinates-based approaches. In *Proc. INFOCOM*, pp. 170–179.

[103] Nichols, J. and M. Claypool (2004). The effects of latency on online Madden NFL football. In *Proc. Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pp. 146–151.

[104] nProtect. GameGuard. web page. *http://global.nprotect.com/product/gg.php*.

[105] Oliveira, M. and T. Henderson (2003). What online gamers really think of the Internet? In *Proc. Network and Systems Support for Games (NetGames)*, pp. 185–193.

[106] Palazzi, C. E., S. Ferretti, S. Cacciaguerra, and M. Roccetti (2004). On maintaining interactivity in event delivery synchronization for mirrored game architectures. In *Proc. Global Communications Conference (GlobeCom)*, pp. 157–165.

[107] Palmer, R. (2008). Implementation and practical evaluation of an anti-cheat protocol for peer-to-peer multiplayer online games. Honours thesis. Department of Computing, Curtin University of Technology.

[108] Pellegrino, J. D. and C. Dovrolis (2003). Bandwidth requirement and state consistency in three multiplayer game architectures. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 52–59.

[109] Pittman, D. and C. GauthierDickey (2007). A measurement study of virtual populations in massively multiplayer online games. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 25–30.

[110] Pittman, D. and C. GauthierDickey (2010). Characterizing virtual populations in massively multiplayer online role-playing games. *Proc. ACM International Multimedia Modelling Conference, Lecture Notes in Computer Science 5916*, 87–97.

[111] Pritchard, M. (2000). How to hurt the hackers: The scoop on Internet cheating and how you can combat it. *http://www.gamasutra.com/features/20000724/pritchard_pfv.htm*.

[112] Qiu, D. and R.Srikant (2004). Modeling and performance analysis of bittorrent-like peer-to-peer networks. In *Proc. SIGCOMM*, pp. 367–378.

[113] Quax, P., P. Monsieurs, W. Lamotte, D. D. Vleeschauwer, and N. Degrande (2004). Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 152–156.

[114] Queeg500 (2009, April). Networking changes on TQ. web page. *http://www.eveonline.com/devblog.asp?a=blog&bid=653*.

[115] Ratnasamy, S., P. Francis, M. Handley, R. Karp, and S. Shenker (2001). A scalable content-addressable network. In *Proc. SIGCOMM*, pp. 161–172.

[116] Ratt (2001, Dec). Showeq open source project. web page. *http://www.showeq.net/*.

[117] Rodolakis, G., S. Siachalou, and L. Georgiadis (2006). Replicated server placement with QoS constraints. *IEEE Trans. on Parallel and Distributed Systems 17*(10), 1151–1162.

[118] Rooney, S., D. Bauer, and R. Deydier (2004, Jan). A federated peer-to-peer network game architecture. Research report, IBM Research GmbH, Zurich Research Laboratory 8803 Ruschlikon Switzerland.

[119] Rowstron, A. and P. Druschel (2001, Nov). Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pp. 329–350.

[120] Seshan, S., J. Pang, and A. Bharambe (2006). Colyseus: A distributed architecture for online multiplayer games. In *Proc. Networked System Design and Implementation (NSDI)*, pp. 155–168.

[121] Shavitt, Y. and E. Shir (2005). DIMES: let the Internet measure itself. *Computer Communication Review 35*(5), 71–74.

[122] Sheldon, N., E. Girard, S. Borg, M. Claypool, and E. Agu (2003). The effect of latency on user performance in Warcraft III. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 3–14.

[123] Siwek, S. E. (2007). Video games in the 21st century: economic contributions of the US entertainment software industry. Technical report, Entertainment Software Association (ESA).

[124] Smed, J., T. Kaukoranta, and H. Hakonen (2002). Aspects of networking in multiplayer computer games. *The Electronic Library 20*(2), 87–97.

[125] Stoica, I., R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM*, pp. 149–160.

[126] Stutzbach, D., R. Rejaie, and S. Sen (2005). Characterizing unstructured overlay topologies in modern P2P file-sharing systems. In *Proc. Internet Measurement Conference (IMC)*, pp. 49–62.

[127] Svoboda, P., W. Karner, and M. Rupp (2007). Traffic analysis and modeling for World of Warcraft. In *Proc. International Conference on Communications(ICC)*, pp. 1612–1617.

[128] Ta, D. N. B., S. Zhou, and H. Shen (2006). Greedy algorithms for client assignment in large-scale distributed virtual environments. In *Proc. Principles of Advanced and Distributed Simulation (PADS)*, pp. 103–110.

[129] Tang, D. T., L. S. Woo, and L. R. Bahl (1978). Optimization of teleprocessing networks with concentrators and multiconnected terminals. *IEEE Trans. Computers C-27*(7), 594–604.

[130] Tarng, P.-Y., K.-T. Chen, and P. Huang (2008). An analysis of WoW players' game hours. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 47–52.

[131] Terdiman, D. (2005, Sep). Virtual goods, real scams. web page. *http://news.zdnet.com/2100-1040_22-144576.html*.

[132] Terdiman, D. (2006, Apr). World of Warcraft battles server problems. web page. *http://news.com.com/World+of+Warcraft+battles+server+problems/2100-1043_3-6063990.html* Last accessed on the 26th June 2006.

[133] Tukey, J. (1977). *Exploratory data analysis*. Addison-Wesley.

[134] Valve (2005). Valve anti-cheat system (vac). web page. *https://support.steampowered.com/kb_article.php?p_faqid=370*.

[135] Valve (2006). Source multiplayer networking. Web page. *http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking*.

[136] Valve (2008, Jan). Survey summary data. Web page. *http://www.steampowered.com/status/survey.html*.

[137] Vivendi (2008). Annual report. http://www.vivendi.com/vivendi/IMG/pdf/20090408_annual_report_en_080409.pdf.

[138] Webb, S. D., W. Lau, and S. Soh (2006). NGS: An application layer network game simulator. In *Proc. Interactive Entertainment (IE)*, pp. 15–22.

[139] Webb, S. D. and S. Soh (2007a). Cheating in networked computer games - a review. In *Proc. Digital Interactive Media in Entertainment and Arts (DIMEA)*, pp. 105–112.

[140] Webb, S. D. and S. Soh (2007b). Round length optimisation for P2P network gaming. In *Proc. Postgraduate Electrical Engineering and Computing Symposium (PEECS)*, pp. 23–28.

[141] Webb, S. D. and S. Soh (2007c). A survey on network game cheats and P2P solutions. *Australian Journal of Intelligent Information Processing Systems 9*(4), 34–43.

[142] Webb, S. D. and S. Soh (2008). Adaptive client to mirrored-server assignment for massively multiplayer online games. In *Proc. Multimedia Computing and Networking (MMCN)*, pp. 6818–17.

[143] Webb, S. D. and S. Soh (2009). Application performance metrics for evaluating delay estimation schemes. In *Proc. Asia-Pacific Conference on Communications (APCC)*, pp. 717–721).

[144] Webb, S. D., S. Soh, and W. Lau (2007a). Enhanced mirrored servers for network games. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 117–122.

[145] Webb, S. D., S. Soh, and W. Lau (2007b). RACS: a referee anti-cheat scheme for P2P gaming. In *Proc. Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pp. 37–42.

[146] Webb, S. D., S. Soh, and J. L. Trahan (2008). Secure referee selection for fair and responsive peer-to-peer gaming. In *Proc. Principles of Advanced and Distributed Simulation (PADS)*, pp. 63–71.

[147] Webb, S. D., S. Soh, and J. L. Trahan (2009). Secure referee selection for fair and responsive peer-to-peer gaming. *SIMULATION: Transactions of The Society for Modeling and Simulation International 85*(9), 608–618.

[148] WoWWiki (2008, Jan). Web page. *http://www.wowwiki.com/.*

[149] Yamamoto, S., Y. Murata, K. Yasumoto, and M. Ito (2005). A distributed event delivery method with load balancing for MMORPG. In *Proc. INFOCOM*, pp. 1–8.

[150] Yan, J. (2003). Security design in online games. In *Proc. Annual Computer Security Applications Conference (ACSAC)*, pp. 286–295.

[151] Yan, J. and B. Randell (2005). A systematic classification of cheating in online games. In *Proc. Network and Systems Support for Games (NetGames)*, pp. 1–9.

[152] Yan, J. and B. Randell (2009). An investigation of cheating in online games. *IEEE Security & Privacy 7*(3), 37–44.

[153] Yeung, S., J. Lui, J. Liu, and J. Yan (2006). Detecting cheaters for multiplayer games: Theory, design and implementation. In *Proc. Consumer Communications and Networking Conference (CCNC)*, pp. 1178–1182.

[154] Yu, A. P. and S. T. Vuong (2005). MOPAR: a mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games. In *Proc. Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pp. 99–104.

[155] Zhang, R., C. Tang, Y. C. Hu, S. Fahmy, and X. Lin (2006). Impact of the inaccuracy of distance prediction algorithms on Internet applications – an analytical and comparative study. In *Proc. INFOCOM*, pp. 1–12.

[156] Zhao, B. Y., L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz (2004, Jan). Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications 22*(1), 41–53.

[157] Zheng, H., E. K. Lua, M. Pias, and T. G. Griffin (2005). Internet routing policies and round-trip-times. *Proc. Passive and Active network Measurement (PAM), Lecture Notes in Computer Science 3431*, 236–250.