

CHEAT-PROOF EVENT ORDERING FOR LARGE-SCALE DISTRIBUTED
MULTIPLAYER GAMES

by

CHRISTOPHER JAY GAUTHIERDICKEY

A DISSERTATION

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

March 2007

CHEAT-PROOF EVENT ORDERING FOR LARGE-SCALE DISTRIBUTED
MULTIPLAYER GAMES

by

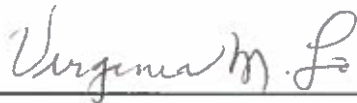
CHRISTOPHER JAY GAUTHIERDICKEY

A DISSERTATION

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

March 2007

“Cheat-proof Event Ordering for Large-Scale Distributed Multiplayer Games,” a dissertation prepared by Christopher Jay GauthierDickey in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:



Dr. Virginia Lo, Chair of the Examining Committee



Date

Committee in charge:

Dr. Virginia Lo, Chair
Dr. Michal Young
Dr. Art Farley
Dr. Greg Landweber
Dr. Wu-chang Feng

Accepted by:



Dean of the Graduate School

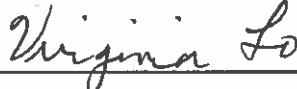
© 2007 Christopher Jay GauthierDickey

An Abstract of the Dissertation of
Christopher Jay GauthierDickey for the degree of Doctor of Philosophy
in the Department of Computer and Information Science

to be taken

March 2007

Title: CHEAT-PROOF EVENT ORDERING FOR LARGE-SCALE
DISTRIBUTED MULTIPLAYER GAMES

Approved: 
Dr. Virginia Lo

Real-time, interactive, multi-user (RIM) applications are networked applications that allow users to collaborate and interact with each other over the Internet for work, education and training, or entertainment purposes. Multiplayer games, distance learning applications, collaborative whiteboards, immersive educational and training simulations, and distributed interactive simulations are examples of these applications. Of these RIM applications, multiplayer games are an important class for research due to their widespread deployment and popularity on the Internet. Research with multiplayer games will have a direct impact on all RIM applications.

While large-scale multiplayer games have typically used a client/server architecture for network communication, we propose using a peer-to-peer architecture to solve the scalability problems inherent in centralized systems. Past research and actual deployments of peer-to-peer networks show that they can scale to millions of users. However, these prior

peer-to-peer networks do not meet the low latency and interactive requirements that multiplayer games need. Indeed, the fundamental problem of maintaining consistency between all nodes in the face of failures, delays, and malicious attacks has to be solved to make a peer-to-peer networks a viable solution.

We propose solving the consistency problem through secure and scalable event ordering. While traditional event ordering requires all-to-all message passing and at least two rounds of communication, we argue that multiplayer games lend themselves naturally to a hierarchical decomposition of their state space so that we can reduce the communication cost of event ordering. We also argue that by using cryptography, a discrete view of time, and majority voting, we can totally order events in a real-time setting. By applying these two concepts, we can scale multiplayer games to millions of players.

We develop our solution in two parts: a cheat-proof and real-time event ordering protocol and a scalable, hierarchical structure that organizes peers in a tree according to their scope of interest in the game. Our work represents the first, complete solution to this problem and we show through both proofs and simulations that our protocols allow the creation of large-scale, peer-to-peer games that are resistant to cheating while maintaining real-time responsiveness in the system.

CURRICULUM VITAE

NAME OF AUTHOR: Christopher Jay GauthierDickey

PLACE OF BIRTH: Laredo, TX

DATE OF BIRTH: June 8, 1971

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
Laredo Community College

DEGREES AWARDED:

Doctor of Philosophy in Computer and Information Science, 2007,
University of Oregon
Master of Science in Computer and Information Science, 2002,
University of Oregon
Bachelor of Science in Computer and Information Science, 2000,
University of Oregon
Associate of Arts in Liberal Arts, 1991, Laredo Community College

AREAS OF SPECIAL INTEREST:

Computer Networks
Peer-to-Peer Networking
Computer Games
Network Security

PROFESSIONAL EXPERIENCE:

Teaching Assistant, Department of Computer and Information Science,
University of Oregon, Eugene, 2005-2006

Graduate Research Fellow, Department of Computer and Information Science,
University of Oregon, Eugene, 2003-2005

Research Assistant, Department of Computer and Information Science,
University of Oregon, Eugene, 2001-2003

Software Developer, Sax Software, Eugene, 1998-1999

Network Administrator, Cegelec ESCA, Seattle, WA, 1996-1997

GRANTS, AWARDS AND HONORS:

Graduate Research Fellowship, National Science Foundation, 2002-2005

PUBLICATIONS:

C. GauthierDickey, V. Lo and D. Zappala, "Using n-trees for scalable event ordering in peer-to-peer games," *Proceedings of NOSSDAV*, June 2005.

V. Lo, D. Zhou, Y. Liu, C. GauthierDickey and J. Li, "Scalable supernode selection in peer-to-peer overlays," *Proceedings of HotP2P*, July 2005.

C. GauthierDickey, D. Zappala and V. Lo, "A fully distributed architecture for massively multiplayer online games," *Proceedings of NetGames*, August 2004.

C. GauthierDickey, D. Zappala, V. Lo and J. Marr, "Low latency cheat-proof event ordering for peer-to-peer games," *Proceedings of NOSSDAV*, June 2004.

D. Zappala, V. Lo and C. GauthierDickey, "The multicast address allocation problem: a theoretical framework and performance evaluation," *Special Issue of Computer Networks*, Volume 45(1), pp. 55-73, May 2004.

V. Lo, D. Zappala, C. GauthierDickey, and T. Singer, "A theoretical framework for the multicast address allocation problem," *Proceedings of Global Internet*, IEEE Globecom, 2002.

D. Zappala, C. GauthierDickey and V. Lo, "Modeling the multicast address allocation problem," *Proceedings of Global Internet*, IEEE Globecom, 2002

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to my advisor, Dr. Virginia Lo, who encouraged me so many summers ago to get involved in research. Her dedication to her students and her keen mind made crafting this dissertation possible. I also wish to acknowledge the faculty in the Department of Computer and Information Science who opened my eyes to a much bigger and more interesting world. Finally, I would like to acknowledge the National Science Foundation who provided me with a Graduate Research Fellowship that made my research possible.

To my wife, Toni, and my kids, Sami and Kyra; without their love and support I would never have been able to accomplish this.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
1.1 Problem Definition	4
1.1.1 Massively Multiplayer Online Games	4
1.1.2 Peer-to-Peer Architecture	6
1.2 Research Challenges	8
1.3 Proposed Approach	10
II. RESEARCH FOUNDATIONS	12
2.1 Introduction	12
2.2 Distributed Systems	12
2.2.1 Distributed Consensus and Event Ordering	13
2.2.2 Distributed Interactive Simulations	15
2.3 Networking Research	15
2.3.1 IP Multicast	15
2.3.2 Reliable Multicast	17
2.3.3 Congestion Controlled Multicast	17
2.3.4 Peer-to-Peer Networking	19
2.4 Multiplayer Game Research	21
2.5 A Taxonomy of Cheats	24
III. THE NEW-EVENT ORDERING PROTOCOL	30
3.1 Introduction	30
3.2 Background and Related Work	32
3.2.1 Terminology	32
3.2.2 Game Protocols	33
3.2.3 Consistency and Event Ordering	34
3.2.4 Time Warp and Rollback	36

Chapter	Page
3.3	The New-Event Ordering (NEO) Protocol 37
3.3.1	The Basic NEO Protocol 37
3.3.2	NEO Consistency 43
3.3.3	NEO Security 50
3.4	NEO Real-time Responsiveness 53
3.4.1	NEO Overhead 54
3.4.2	NEO with Pipelined Rounds 55
3.5	Performance Experiments 56
3.5.1	Methodology and Metrics 58
3.5.2	NEO and Lockstep 59
3.5.3	NEO Group Performance 65
3.5.4	How Well Does NEO Scale? 70
3.6	Conclusion 73
IV.	N-TREES 76
4.1	Introduction 76
4.2	Related Work 78
4.3	Definitions 80
4.4	The N-Tree Protocol 81
4.4.1	Node Leader Selection 83
4.4.2	Event Propagation 83
4.4.3	Forecasting 85
4.4.4	Event Ordering and Majority Voting in the N-Tree 87
4.4.5	Bootstrapping 87
4.4.6	Joining the N-Tree 90
4.4.7	Leaving the N-Tree 90
4.4.8	Subdivision of Leaves and Collapsing of Branches 91
4.5	N-Trees Consistency 91
4.6	N-Trees Security 94
4.7	Analysis of N-Trees 96
4.7.1	Asymptotic Analysis 97
4.7.2	Performance Analysis with Local Event Propagation 99
4.8	Measurements of Virtual Populations in MMOGs 101
4.8.1	Background 103
4.8.2	Measurement Technique 104
4.8.3	Measurement Results 107
4.8.4	Measurement Summary 116

Chapter	Page
4.9 Simulation Experiments	117
4.9.1 Modeling a Virtual World	117
4.9.2 Methodology	119
4.9.3 Experimental Results	121
4.10 Conclusions	125
V. CONCLUSIONS AND FUTURE WORK	128
BIBLIOGRAPHY	134

LIST OF FIGURES

Figure		Page
3.1	NEO rounds	39
3.2	NEO update with voting	40
3.3	Pipelining rounds in NEO	57
3.4	NEO simulation topology	58
3.5	Playout latency of NEO and Lockstep	60
3.6	Update frequency of NEO and Lockstep	61
3.7	Update frequency of NEO and Lockstep with packet loss	63
3.8	Playout latency of NEO and Lockstep with packet loss	64
3.9	Update frequency of various NEO group sizes with packet loss	67
3.10	Playout latency for various NEO group sizes and packet loss	67
3.11	Rollback required as NEO group size increases	69
3.12	Length of rollback as NEO group sizes increase	69
3.13	Scalability of NEO with 10 updates/second	72
3.14	Scalability of NEO with 5 updates/second	73
4.1	Cartesian application state space and quadtree representation	82
4.2	Event propagation in N-Trees	84
4.3	N-Tree voting	88
4.4	Mapping virtual world coordinates into CAN	89
4.5	N-Tree consistency	93
4.6	Latency for local events with N-Trees, MCast Regions, and Mercury	101
4.7	Virtual population over time	108
4.8	CDF of play duration	110
4.9	Time versus playing duration	110
4.10	Session durations based on friends list	111
4.11	CDF of session durations based on friends list	111
4.12	Arrival rate over time	112
4.13	Arrival rate of players per minute	113
4.14	Distribution of players per zone	114
4.15	CDF of player distribution in zones	115
4.16	Histogram of players versus the number of unique zones visited	115
4.17	Histogram of players versus time spent in a zone	116
4.18	Visualization of N-Tree simulator	120
4.19	Average path lengths in the N-Tree	122
4.20	Histogram of path lengths for 1,000, 10,000, and 100,000 peers	123

Figure	Page
4.21 Average forecasting time	124
4.22 Maximum path lengths in the N-Tree	125
4.23 Maximum forecasting time	126

LIST OF TABLES

Table		Page
1.1	Archetypes for multiplayer games	9
2.1	Features of DHTs	21
2.2	A taxonomy of cheating	25
3.1	Contributions of game protocols	31
3.2	Vote table for a NEO player	41
3.3	Playable parameters for games	54
3.4	Experiment parameters for NEO with increasing delay	60
3.5	Experiment parameters for NEO with packet loss	62
3.6	Experiment Parameters for varying NEO Group Sizes	66
4.1	Asymptotic messaging costs	98
4.2	Experiment parameters for N-Tree simulations	121

CHAPTER I

INTRODUCTION

Since the advent of the Internet, people have sought to collaborate and interact over long distances on the network for work, education, training, and entertainment. Encompassing these goals are a set of applications we call real-time, interactive, multi-user (RIM) applications which bring people together over the Internet for a variety of reasons. Multiplayer games, distance learning, collaborative whiteboards, immersive educational and training simulations, and distributed interactive simulations are examples of RIM applications.

Of these RIM applications, multiplayer computer games are an important area for research for several reasons. First, the game industry is a multi-billion dollar industry which has an annual gross income larger than Hollywood at the box-office. This widespread market penetration makes game research relevant to the average person and to the economy in general. Second, massively multiplayer online games (MMOGs) have thousands of players, require complex computational resources to simulate a large-scale virtual world, and are the largest interactive environments in existence. As extensions of distributed computing, the scale and complexity of MMOGs presents a difficult and interesting research challenge. Third, as part of the class of RIM applications, research in MMOGs directly benefits other RIM applications. As a research topic, MMOGs provide a rich area of study and experimentation which will have a direct impact on both computer science and our society.

Traditionally, MMOGs have been designed with a client/server communication architecture. This architecture has the advantage that a single authority orders events, resolves conflicts in the simulation, acts as a central repository for data, and is easy to secure. On the

other hand, this architecture has several disadvantages. First, the computational complexity of simulating a large scale virtual world for tens of thousands of players requires monolithic systems to be greatly over-provisioned to meet peak demands while costly resources lie idle the majority of the time. For example, a typical EverQuest world [1], which is simulated on a cluster of servers, can handle around 2500 players concurrently [2]. Yet, with 150,000 players online at any time, approximately 1500 servers are required worldwide, averaging only 100 players per server [2]. Second, the client/server architecture introduces additional delay on messages between players because they are always forwarded through the server. Third, traffic at the server increases with the number of players, creating localized congestion and high-bandwidth requirements at the server's location. One local game developer stated that the bandwidth requirement of their MMOG was equivalent to all of the local telephone bandwidth in the city combined.

To address the limitations of the client/server architecture, researchers have turned to peer-to-peer architectures for MMOGs [3, 4, 5, 6]. The peer-to-peer architecture allows peers to send messages directly to each other, reducing the delay for messages and eliminating localized congestion. It allows players to start their own games without the incredible investment in resources required by the client/server architecture. Furthermore, this architecture allows games to overcome the bottleneck of server-only computation by harnessing the processing power and storage capacity of the players' machines. Available resources grow organically as more peers join the system. Finally, this architecture is more resilient and available because it does not have a single point of failure.

The realization of a large scale peer-to-peer architecture for MMOGs, however, faces a number of clear challenges. Any peer-to-peer communication architecture for MMOGs must maintain a consistent view of the game among peers. In a small-scale peer-to-peer architecture, consistency can be achieved through a distributed event ordering protocol which ensures that all players agree on the same set of events. However, MMOGs have a real-time constraint on message passing because they are interactive by design. This constraint typically falls between 100ms and 250ms, depending on the type of game. Thus, events must be able to be distributed and ordered within this tight time constraint.

Adding more of a challenge is the fact that the event ordering protocol must be both cheat-proof and scalable for MMOGs. Game players have a long history of cheating to

gain unfair advantages and can modify the flow of packets between players to cheat without being detected or prevented by Byzantine agreement protocols [7]. Furthermore, cheat-prevention and indeed Byzantine agreement protocols typically require all-to-all communication and therefore do not scale well.

In this dissertation, we address the problem of cheat-proof and scalable event ordering with real-time constraints for MMOGs under a peer-to-peer architecture. Research to date has only addressed the requirements of this open problem individually, e.g. event ordering protocols have been developed that prevent some cheats but which suffer from long latencies and/or lack of scalability; or scalable peer-to-peer protocols that are highly vulnerable to cheating. In addition, only a few types of cheating have been simultaneously addressed within current protocols.

Our work is the first to synthesize these requirements into a single communication architecture. This approach is necessary due to the complex requirements. We cannot simply re-engineer existing protocols to solve these problems, but instead our protocols must be initially designed to meet all of these requirements.

Accordingly, we have created two protocols, the New-Event Ordering (NEO) protocol and the N-Tree protocol which address real-time event ordering, cheating, and scalability. NEO uses a novel voting mechanism that can achieve consistency between peers in real-time under a variety of network conditions. N-Trees build a tree based on the virtual world and the distribution of the population within it for event ordering and propagation. It leverages NEO at the leaves to provide fast event ordering and *forecasts* updates to prevent rollback.

In addition, we also formalize consistency with respect to games and expand upon distributed system ideas of consistency. Both NEO and N-Trees provide a type of *majority game consistency*, which is similar to the concept of Lamport's *sequential consistency* [8], except that only the subset of updates which were voted on by a majority of peers in the game are accepted. This concept of consistency allows NEO and N-Trees to mask both lost and late packets and continue ordering events without requiring all updates to arrive.

1.1 Problem Definition

1.1.1 Massively Multiplayer Online Games

A massively multiplayer online game (MMOG) is a computer game in which many players interact within a virtual world over the Internet. The number of concurrent players is typically on the order of 10^4 or more. We define an MMOG as being *scalable* if it can support 10^4 or more players and the cost in messaging between players is $O(n)$, where n is the number of players in the game.

MMOGs also have persistent state. This means that an MMOG, unlike other networked games which end after some goal is completed, can continue indefinitely. Players join the game and play until they are ready to quit, at which point the state of their alter-ego in the game is saved. When they return, the state is restored. This also holds true for the virtual world. For example, a player might own a house in an MMOG that other players can visit even when she is not online. While many players often play in accordance with the established rules of the game, MMOGs are also plagued by players who cheat to gain advantages for themselves.

Games typically fall into three different archetypes known as *first-person shooters* (FPS), *role-playing games* (RPG), and *real-time strategy games* (RTS):

- *First-Person Shooter (FPS)*: In an FPS, the main goal is typically to kill other players with various weapons in the virtual world, hence their name. Because a player can only sustain several hits, depending on the weapon, fast reflexes and reaction times play a significant role in the success of a player. Note that other games, such as racing games, also require fast reflexes and reactions, and therefore fall into this category. The *twich* nature of FPSs require a low round-trip network latency around 100ms.
- *Role-Playing Game (RPG)*: In an RPG, one of the primary goals in the game is to develop one's alter-ego by increasing abilities, skills, and gathering new equipment. These goals are achieved by banding together with other players to explore new lands, kill monsters, and sometimes by fighting with other players. However, combat in RPGs is resolved through a mathematical system based on the alter-ego's

abilities and equipment, and not entirely through the reflexes of the player. Thus, players can tolerate a slower response time from the game.

- *Real-Time Strategy (RTS)*: In an RTS, a player is in control of *units* in a virtual world, where a unit might be a soldier, vehicle, or building. The player acts as the commander of the units and instructs them on what actions to take. Players compete with each other to destroy all of the units of another player, or to capture some vital unit or location in a game. Combat is resolved through strategy and the abilities of the units, and not through reflexes. Thus, players have been reported to tolerate latencies of up to one second, without it detracting from the game [9].

The virtual world of commercial MMOGs are instantiated as *realms*, which are copies of the virtual world that differ in player population and current state. Realms are geographically distributed across the globe so that US players connect to US realms, European players connect to European realms, and so forth. As an analogy, consider the game Monopoly™. At any time, millions of people might be playing Monopoly, but each game can support a maximum number of players. While each game has the same pieces and cards, the players and state of each game are different.

A realm in an MMOG can handle several thousand players simultaneously. For example, in the popular MMOG called EverQuest [1], a typical cluster handles around 2500 players concurrently and approximately 10,000 registered players [2]. To compensate for millions of subscribers, game companies host hundreds of realms across the globe.

Each realm is further divided into *zones*, or geographically distinct areas in the virtual world. Each realm contains all the same zones with the same computer controlled inhabitants (though some of the population may be randomly generated). For example, *World of Warcraft* [10] has approximately 70 zones with names such as Elwynn Forest and Westfall. Players within the same realm can move freely between zones.

Throughout the game, players generate moves, also referred to as events or updates, which must be communicated to some or all of the other players. Associated with each move is a timestamp that allows players to order events and maintain a consistent view of the virtual world. Due to network conditions, moves sometimes arrive late, causing players to *rollback* prior moves to insert the late move in their sequence of events. Another

technique, called *dead-reckoning* is used to predict where a player might move to when her update has not been received.

The current commercial state of the art in MMOG design uses a client/server communication architecture. In this architecture, players (clients) generate moves and send them to a centralized server. The server timestamps moves as they arrive and then appropriately disseminates moves to the players according to whether or not a player is within the scope of a move.

The client/server architecture has several failings when used as the primary architecture for MMOGs. First, hosting an MMOG requires a huge investment in resources including network bandwidth and server clusters. MMOGs are geographically distributed so hosting requires server locations across the globe. Second, the server is a single point of failure so that if one server fails, thousands of players are unable to play. Third, servers must be over-provisioned to handle peak crowds. Typically more players are able to play during non-working hours and weekends implying that servers are under-utilized during non-peak hours. Fourth, each realm is limited in scope by the computational power of the server—thus if a game requires more computational power, more servers must be added. Finally, the client/server architecture introduces increased latency because all messages must pass through the server before reaching other players.

1.1.2 Peer-to-Peer Architecture

Peer-to-peer (P2P) architectures for MMOGs were recently proposed to overcome the limitations of the client/server architecture. Peer-to-peer architectures have been shown to be scale-free for certain classes of applications, such as distributed searching and storage [11, 12, 13, 14, 4]. Thus, with more players we gain more resources for running the MMOG, including increased storage capacity and computational power. A peer-to-peer architecture allows peers to send messages directly to each other, thereby reducing the latency for updates. In addition, the peer-to-peer architecture gains resilience because the failure of one or more peers does not cause the failure of the entire system.

In a peer-to-peer communication architecture, players must timestamp their own events¹ and then broadcast those events to other players, using rollback if necessary for events that arrive late from other players. However, additional problems are introduced through this method. First, if every player broadcasts their update to every other player, the peer-to-peer architecture will not scale with the number of players. Second, because players are responsible for timestamping and distributing their own moves, they are also capable of cheating in several ways by either faking timestamps, purposely dropping updates, or purposely delaying updates to other players. Third, it is necessary to maintain consistency in a distributed fashion unlike the server which was the ultimate authority on the state of the system.

Current protocols developed in the research community address one of three problems: low-latency event delivery, cheat-prevention, or scalability. Diot and Gautier's work on *bucket synchronization* address low-latency event delivery by forcing a more granular view of time so that fewer events have to be rolled back [15], but does not address cheating or scalability. Baughman and Levine address cheat-prevention with the Lockstep protocol by forcing all players to commit moves, agree on those moves, and then reveal the moves [3]. They then attempt to address scalability, but do so by statically subdividing the virtual world into distinct areas. However, their protocol is subject to certain protocol cheats and does not address low-latency event delivery. Bharambe et al. and Knutsson et al. use distributed hash tables (DHT) to address scalability but ignore the problem of cheating or low-latency event delivery [4, 6].

Our goal to design a peer-to-peer game architecture using a holistic approach. Thus, we will develop a peer-to-peer architecture that provides low-latency event delivery, while being scalable and cheat-proof. When these problems are considered separately, as researchers have done in the past, very different solutions arise, and trying to subsequently address the missing elements is a non-trivial task at best.

¹Note that the alternative, where a player timestamps an event when it is received, will lead to inconsistent state between players.

1.2 Research Challenges

In order to provide cheat-proof and real-time event ordering for MMOGs, we must solve four key problems. These problems are:

1. *Latency*: Message passing between players must have very low latency in order to meet the real-time constraints required by interactive games.
2. *Consistency and Event Ordering*: Because we are using a peer-to-peer architecture, the state in the system will quickly become inconsistent without a protocol to maintain consistency.
3. *Scalability and event notification*: All to all message passing is not a scalable method for event dissemination and agreement. In particular, we must be able to determine who needs to receive events so that every peer does not need to be contacted with every generated event.
4. *Cheating*: Historically, players find methods to cheat such as through the modification of individual packets or packet flows. A peer-to-peer game architecture must combat a wide range of cheats for the game to appeal to players.

With respect to latency, multiplayer games must be able to exchange messages within a small time limit in order to maintain their interactive nature. This time limit is based on the delays that players of a given multiplayer game archetype can tolerate before they find the game unplayable. We list the maximum tolerable delays and archetypes in Table 1.1 [16]. The first two categories, first-person shooters and sports and racing games, have the lowest tolerance for latency. Players with round-trip time delays above those listed in Table 1.1 perform worse than players with shorter latencies. The third category, role-playing games, make up the bulk of large-scale multiplayer games on the Internet and players tolerate round-trip times of up to .5 to 1 second. The fourth category of multiplayer games are real-time strategy games, in which players tolerate round-trip times of up to 1 to 1.5 seconds.

Second, our peer-to-peer architecture needs to provide consistency guarantees so that we can ensure the peer-to-peer system will work correctly. We can provide consistency

TABLE 1.1: *Archetypes for multiplayer games: Maximum tolerated round-trip times and examples of large-scale multiplayer games.*

Archetype	Maximum Tolerated RTT	Example MMOG
First-person Shooter (FPS)	150–250 <i>ms</i>	Planetside
Sports and Racing games (SRG)	150–500 <i>ms</i>	<i>None developed</i>
Role-playing Game (RPG)	500–1000 <i>ms</i>	Everquest
Real-time Strategy (RTS)	1–1.5 <i>s</i>	<i>None developed</i>

through an event ordering protocol, but our protocol must be able to operate over a best-effort network such as the Internet. In other words, it must tolerate high latencies, unreliable peers, and dropped packets.

The third problem we must consider is scalability and event notification. If we provide an event ordering protocol, we must be able to scale it with the number of users in the system. Traditional event ordering, such as the Paxos protocol, the Isis system, or RT-Cast [17, 18, 19], are not options because they do not scale with the number of users due to all-to-all message passing.

The event ordering protocol must be able to quickly determine which users need to receive a particular event. Clearly the architecture is not scalable if a large number of peers must be contacted in the process of disseminating an event to its intended recipients. Multicast could be used to reduce the communication costs so that fewer messages are delivered [20, 21, 22, 23, 24, 14]. However, multicast trees are built using the shortest path between members of the tree. While this reduces the delay from root to leaf, an event may have to traverse several (possibly end-system) nodes before reaching its destination. Thus, we believe a new solution is needed.

Finally, we must address the problem of cheating. Historically, some players are willing to cheat and gain an unfair advantage over honest players. Cheating can occur at many levels in the game, from breaking game rules to launching denial-of-service attacks against opponents. An architecture can only address certain kinds of cheats—in particular, those that occur at the protocol level. However, due to the probability that players will cheat if they can, any architecture that does not address cheating will not be considered a valid alternative for the client/server architecture.

We must address all four problems in an integrated manner to solve the problem of cheat-proof and real-time event ordering for MMOGs. In the past, researchers have considered these problems in isolation [15, 3, 4, 6]. However, adapting their architectures to solve all of these problems is a difficult task at best and in fact is still an open research problem. On the other hand, we will address latency, consistency and event ordering, scalability and event notification, and cheating together in the design of a scalable peer-to-peer architecture for MMOGs.

1.3 Proposed Approach

Our work is the first to take a holistic approach towards addressing the requirements of scalability, protection against cheating, and real-time responsiveness. This integrated approach is necessitated by the complex interactions among these requirements and by the fact any solution that fails to achieve minimum performance standards for all three requirements simultaneously is not a viable solution in the highly competitive world of MMOGs.

Our approach to the problem of cheat-proof event ordering for large scale MMOGs consists of two inter-related protocols. The NEO (New Event Ordering Protocol) is designed for smaller groups of players. NEO uses a novel majority voting system that can achieve event ordering in the presence of late messages, lost messages, and cheating. Strict deadlines on message passing allow us to totally order events and prevent common protocol-level cheats. We present NEO in Chapter III.

Scalability is achieved through the N-Trees protocol which hierarchically organizes peers by their application level interests so that a peer mostly needs to communicate and order events with peers that are close by. N-Trees are formed by recursively subdividing and N-dimensional space evenly along each dimension. Peers are then placed in leaves of the N-Tree according to their scope of interest in the game, using NEO to totally order events at the leaves. Peers that are close by in the virtual world will be close by in the N-Tree, thus events will only be ordered with nearby peers. Global events are handled by a technique we develop called *forecasting* which propagates messages to the relevant NEO groups in a timely and cheatproof manner. We present N-Trees in Chapter IV.

In this dissertation, we also formalize the notion of consistency and event ordering for large scale MMOGs by extending the classic notions from distributed computing to the domain of large scale peer-to-peer applications. We define several notions of game consistency, and show how these relate to safety and liveness of game protocols.

This dissertation will have a broad impact on future research in MMOGs, RIM applications, and distributed systems. The immediate impact of our work will be to help large-scale peer-to-peer games become a viable alternative to the client/server architecture. It will stimulate research in this area to look at other unsolved problems, such as cheat-proof scheduling of peer-to-peer computations and large-scale, cheat-proof peer-to-peer storage for MMOGs. The medium range impact will be on RIM applications including distance learning, collaborative virtual environments, immersive educational and training simulations, and distributed interactive simulations. Because the requirements for RIM applications are a subset of the requirements for MMOGs, our work will directly benefit research in these areas. Finally, the long range impact of our work is on distributed and peer-to-peer computing and networking, particularly in the fundamental areas of scalable event ordering and cheat-prevention. Our notions of consistency may be applied to other distributed systems for scalability purposes. Furthermore, those distributed and peer-to-peer systems with an application state space that can be hierarchically subdivided and with events that can be meaningfully scoped will benefit from the solutions we propose for MMOGs.

The rest of this dissertation is organized as follows: Chapter II explains the common terms used in networking and game research. It also explains the common cheats and presents the background material that forms the foundation of our research. Chapter III then explains the NEO protocol, proves that it is resilient to protocol-level cheating, and uses simulation to show that NEO can send a sufficient number of updates for the common game archetypes under a variety of network conditions. By maintaining consistency, sending sufficient updates, and being cheat-proof, NEO is a viable protocol for peer-to-peer multiplayer games. Chapter IV presents N-Trees as a structure for organizing peers into large-scale, multiplayer games. We analyze the theoretical performance of N-Trees and prove that they provide consistency for multiplayer games. We perform a measurement study to develop a more realistic model for simulation and then we study N-Trees experimentally through simulations of up to 100,000 players. We conclude in Chapter V.

CHAPTER II

RESEARCH FOUNDATIONS

2.1 Introduction

Our research is related to three areas: distributed systems, networking and computer games. Therefore, a fundamental knowledge in these areas is required to understand our protocols and choices in our designs. In the context of distributed systems, we discuss the distributed consensus problem, event ordering and consistency in Section 2.2. Our research builds on these theoretical foundations, expanding the notions of event ordering and consistency to apply them towards distributed MMOGs.

Within the field of networking, multicast and peer-to-peer networks are key areas that form the basis for our research with NEO and N-Trees as described in Section 2.3. We adopt the basic ideas behind multicast and distributed hash tables to provide a scalable event ordering architecture.

We discuss modern research in computer games in Section 2.4, and explain how prior research has looked at event ordering and simplified cheating models. An important common theme throughout game research is the problem of cheating. We define a taxonomy of cheating in Section 2.5, necessary to understanding our research with NEO and N-Trees.

2.2 Distributed Systems

A distributed system is defined as one in which the processes only coordinate through message passing [25]. This definition implies that hardware will not be used to synchro-

nize processes or maintain consistency. In our research, we are primarily interested in distributed systems that communicate over the Internet and therefore we do not consider hardware mechanisms or make assumptions about the reliability or capabilities of the network.

Distributed systems are divided into two models: synchronous and asynchronous. In the synchronous model, process execution either occurs in synchronous rounds (i.e., they proceed in lockstep), or events can be timestamped according to a global clock [26] by using a clock synchronization protocol such as the Network Time Protocol (NTP) [27].

NTP is a client/server protocol where servers are arranged in a tiered hierarchy. A client synchronizes its clock with a server by measuring the round-trip time between the two systems and estimating the difference between the current clocks each system has. Depending on the level in the hierarchy (tier 1 NTP servers are more accurate than tier 2 servers, which are more accurate than tier 3 servers), a client can accurately synchronize clocks to within milliseconds.

The advantage of the synchronous model is that it is easier to reason about, with the caveat that most real distributed systems are not completely synchronous. In the asynchronous model, processes execute local instructions at arbitrary speeds. This model has the advantage that algorithms designed for it can run on *all* types of networks without timing guarantees. The disadvantage of the asynchronous model is that some problems are more difficult, if not impossible, in the asynchronous system [26]. For example, the Byzantine Agreement problem is impossible under an asynchronous system model. Therefore, we only consider synchronous models for computer games.

2.2.1 Distributed Consensus and Event Ordering

An important problem in distributed systems is the need to agree on things in the system such as the current state of a variable or the result of an algorithm. In the *distributed consensus* problem, processes in a distributed system propose a value and attempt to reach agreement on that value. Distributed consensus algorithms guarantee that all non-faulty processes will eventually reach agreement. The important results from past research show that in an asynchronous system, distributed consensus is impossible while in a synchronous

system, distributed consensus can be achieved by $3f + 1$ processes in f rounds if only f failures of any kind occur [7]. However, if we add digital signature to prevent forgeries, $f + 2$ processes can reach consensus given f failures in f rounds of communication [7].

Event ordering is similar to distributed consensus in that all non-faulty processes are trying to agree on the time when each event occurred [28]. The result is a total ordering of events in the system. We can further distinguish between *strong* and *weak* event ordering:

- *Strong Event Ordering*: All events follow the same order on all nodes of the distributed system. This total ordering corresponds to Lamport's *sequential consistency* [8].
- *Weak Event Ordering*: An ordering of events in which some systems may see a different ordering, but the shared state of all systems remains consistent [28]. Consistency is achieved through distributed synchronization mechanisms such as locks and critical sections.

In a distributed system that uses message passing to alter the shared state of the system, for n participants, any event ordering algorithm requires at least $\Omega(n^2)$ messages between all participants sharing the state, and at least 2 rounds of communication before consensus can be reached [29]. The Paxos algorithm, which only tolerates stopping failures, requires up to five rounds of communication and a majority of nodes must have reliable communication with the leader [17]. Early systems, such as Isis [18] and Orca [30], do not scale because they assume the distributed system is tightly coupled over a local area network.

Unfortunately, the time-sensitive messaging requirements for distributed games to use a byzantine agreement protocol make it impractical. A typical game needs to exchange around 10 updates per second over a best-effort network. Using a byzantine agreement protocol (even with digital signatures) would require on the order of $O(n^2)$ messages per player for n players each round.

We build upon the concepts of event ordering and consistency in distributed systems in the development and analysis of our protocols.

2.2.2 Distributed Interactive Simulations

Distributed Interactive Simulations (DIS), which are typically training simulations executed on a distributed system for the purpose of simulating battlefield scenarios and training soldiers in synthetic environment, were some of the first RIM applications. SIMNET comprised the first attempt at large scale, distributed simulations, followed by DIS, was funded by DARPA in the late 1980s and early 1990s [31, 32, 33].

While DIS was designed to be a distributed system, the resulting protocols were not general enough for all RIM applications and had the requirement that the underlying network support multicast and guarantee latency within about 200ms. Using a combination of supercomputers, high-speed networks and highly optimized DIS protocols, the latest DIS experiments supported 100,000 entities. DIS research does not consider security, except to encrypt packets when simulations are top-secret.

In our research, we propose addressing the problem of scalability through peer-to-peer networking—a significantly different approach than DIS uses. Furthermore, security against cheats is of paramount importance.

2.3 Networking Research

Communication protocols and paradigms are relevant to our research because we build on the ideas from multicast and peer-to-peer networking in the development of our protocols. Thus, we explain how multicast works, its development into application layer multicast, and the history and workings of peer-to-peer networks.

2.3.1 IP Multicast

IP multicast is a form of group communication that is implemented at the IP level in the network. Participants join a *multicast group*, which acts as a logical address (and is in fact assigned a special multicast IP address), and are able to send and receive packets with that address. IP multicast forms a multicast tree to disseminate packets over, where routers in the Internet act as nodes in the tree and duplicate packets when they need to be sent over multiple branches.

The original concepts behind multicast were formalized by Deering and Cheriton in [20]. The goal of their design was to make multicast as similar to unicast as possible. To summarize their design, multicast uses groups that can be addressed by a single address, groups are open (knowledge of group membership is not necessary), hosts can join and leave at will, and hosts can belong to more than one group at a time. The first large scale multicast sessions began with the MBone, a set of tunnels over the Internet that connected to LANs that were multicast capable [34].

The primary problem with the original multicast routing protocol (or DVMRP) is that it is efficient only if a large number of nodes in the system are participating, due to the amount of message passing that must occur to maintain multicast trees [20]. This led to research that focused on *sparse* protocols, where membership in the system only comprises a small percentage of the total number of nodes. Protocol Independent Multicast (PIM) [35] and Core Based Trees [36] are examples of sparse multicast protocols.

In building multicast trees, two primary methods have been investigated: shortest path trees and core based trees. Shortest path trees are rooted at the source with receivers at the leaves. Core based trees pick a central node in a minimum spanning tree that acts as the core. Sources unicast messages to the core which are then multicast to all group members. Core placement was studied extensively in [37, 38].

Unfortunately PIM and CBT did not address several fundamental problems. First, neither considered inter-domain multicast, which is important since ISPs wish to do policy-based routing so that they can direct as much of their traffic over their own networks as possible to reduce cost. The MASC/BGMP architecture addressed this by allowing intra-domain multicast to use whatever multicast protocol it deemed necessary and by building a bi-directional core-based tree rooted in the domain of the source of the multicast group [39].

Second, some researchers felt that the multicast model was too general. Thus, single-source multicast (SSM) was created in the form of EXPRESS multicast [40]. In SSM, only one source can multicast messages to the group and an SPT is built rooted at the source. Holbrook and Cheriton argued that SSM was applicable to many multicast problems such as multimedia broadcast [40]. Zappala and Fabbri added proxies to SSM in order to extend the SSM model to the general multicast model; in other words, proxies allow any number of sources to send over the SSM tree with receivers subscribing to additional sources [41].

The use of multicast in distributed games could significantly reduce network traffic. In our research, N-Trees act as a multicast tree for disseminating updates to interested players. In addition, by solving the problem of totally ordered, reliable multicast, we also solve the distributed consensus problem [25].

2.3.2 Reliable Multicast

IP multicast assumes best-effort delivery and does not guarantee all packets will arrive at the receivers. Various researchers have designed reliable multicast variants [42, 43, 44, 45, 46, 47]. These designs typically involve using some nodes to cache data that can then later be recovered. In addition, reliable multicast requires that receivers can actually detect lost packets. This implies that packets are either sequenced or that receivers expect packets periodically and can therefore detect missing packets.

In LBRM, a special logging server is added to the network and records all multicast packets. The logging servers provide packet recovery for reliability since native multicast uses unreliable transport¹ [42]. RMTP is similar to LBRM in that certain receivers are designated to cache packets. In SRM, however, any receiver can respond to a lost packet message [43, 45]. RMX uses a designated host at the LAN level and the multicast tree is built over the RMX receivers [46]. Each RMX node limits the data flow according to the reception abilities of its receivers. Finally, Kasera et al. propose using multiple multicast groups for error recovery [47]. In their proposition, nodes subscribe to alternate channels to recover missing packets.

2.3.3 Congestion Controlled Multicast

In addition to reliability, scientists have also studied congestion control with multicast. Congestion control is vital to the sustainability of the Internet as Jacobson described in [48]. In order to provide congestion control for multicast effectively, two problems need to be addressed. First, we have to make sure to only count one packet as lost when a single packet is lost over multiple paths. Second, we have to adjust the sending rate based on

¹LBRM was designed in the context of distributed interactive simulations, where adding logging servers to the system was not considered a serious issue.

the congestion window for *all* receivers [49, 50]. This means that a source needs to track the congestion window for each receiver individually in order to keep performance at a maximum without overwhelming any single user.

RLM addresses the congestion issue by assuming that we can divide a multicast stream into multiple layers [51]. For instance, a media stream can be layered, with each successive stream adding higher quality data to the receiver. MTCP, on the other hand, uses specific receivers to collect data about their children in the multicast tree [52]. This data is collected and collated up the tree until the source receives a report that represents the state of the multicast tree. The source then adjusts the sending rate based on the report.

Application Layer Multicast

While a large amount of research went into designing multicast, one question is why it has not been widely deployed. First, ISPs have been leery to enable new technology that has not been standardized (no one has agreed on the *best* form of multicast). Second, billing in the multicast model is a difficult problem. A single source can send just one packet that is duplicated thousands of times in another domain. Thus, one ISP would use very little resources while generating a large volume of traffic in another.

By building multicast at the application layer, instead of at the network layer, researchers hope to make multicast available on the Internet. They also argue that the network layer is not an appropriate place for multicast when one considers that multicast requires state to be kept on each router (which is the part of the end-to-end principle [53]).

In order to design application layer multicast, an overlay on top of the physical topology must be built. Overlays are either structured or unstructured. Structured overlays are built by algorithms that maintain a particular structure, such as a hypercube, while unstructured overlays are designed to maintain other network properties such as connectivity or low graph diameter.

Using the overlay, application layer multicast builds a tree for message distribution. Narada was one of the first application layer multicast protocols [21] and was shown to be capable of handling media streaming [13]. Narada works by building a mesh between members of a group and then building a tree for message distribution. SCRIBE, on the

other hand, uses Pastry [54] to build a peer-to-peer network with the group members [14]. Each multicast group maps to a particular node in the overlay which then acts as the root of the multicast tree. We also note that several other application layer multicast protocols have been designed, and these two are simply representatives of the research in application layer multicast.

2.3.4 Peer-to-Peer Networking

Peer-to-peer networking is a fully distributed form of storage and retrieval over a network that has been used successfully recently and is currently the subject of a large amount of research. With peer-to-peer networking, peers act as both clients and servers on the network and form an overlay topology for routing purposes.

Unstructured Peer-to-Peer Networks

Perhaps the most notorious peer-to-peer networking system most people are familiar with is Napster, which gained its notoriety from its users illegally sharing music files [55]. Though users transferred files directly from each other, Napster was not entirely peer-to-peer. Instead, Napster used a central server that clients connected to in order to locate music stored by other peers and peers transferred files directly to each other.

Gnutella, on the other hand, is a completely peer-to-peer network for file-sharing [56]. Gnutella has a two-level hierarchy, composed of peers and *ultrapeers*. Ultrapeers are reachable by all peers and are well-provisioned nodes in the network. Peers connect to ultrapeers initially and all queries for files are sent to ultrapeers. Ultrapeers index the data stored by peers to reduce the number of peers that have to be contacted directly.

Gnutella's file searching protocol is a simple flooding protocol which floods a query from one ultrapeer to other ultrapeers with a time-to-live (TTL) field that is decremented by one for each new ultrapeer it is forwarded to. The TTL field is set dynamically based on an approximation of how many peers an ultrapeer calculates will need to be contacted to locate the file. If the query fails, the TTL is increased and the query is repeated.

The main drawback of unstructured peer-to-peer networks is that if we need to locate data on the network, we typically have to flood our requests to a large number of peers.

Gnutella's hierarchy reduces this cost significantly and measurement studies have shown Gnutella to have over 1 million peers connected simultaneously.

The success of Napster, and other so called *unstructured* peer-to-peer networks, has prompted the creation of other peer-to-peer systems and motivated the research community to explore the viability of peer-to-peer storage.

Structured Peer-to-Peer Networks

Structured peer-to-peer networks enforce a structure on the overlay network in order to guarantee properties of the network, such as logarithmic routing times. One class of structured peer-to-peer networks, called Distributed Hash Tables (DHTs), work similarly to unstructured peer-to-peer networks in that they both have the ability to store and lookup information on the network. However, structured peer-to-peer networks organize peers according to some structure so that lookup operations are guaranteed to find the data on the network if it still exists.

DHTs work similarly to hash tables. They have a *key space*, which is some number of bits in length, that maps to the *value space*. DHTs typically have two functions: *store(key, value)*, and *lookup(key)*. Objects that are to be placed in the network use some kind of cryptographically secure hashing function to return a hash value that fits in the key space. Nodes in the peer-to-peer network are also hashed into the key space. Thus, to determine where to store an item on the network, one simply hashes the object and routes to the node whose ID is closest to the object.

In order to route objects or requests to other nodes in the network, peers need to maintain routing tables. This maintenance is dependent on the DHT algorithms, but typically the DHT forms some sort of logical structure. Chord [12], for example, is a logical ring, while CAN is a d -dimensional torus [11]. The differences between route management in these DHTs are in routing guarantees (how long it takes to route to an item) and network locality (how well does the overlay map to the underlying network). Table 2.1 lists the differences between some of the common DHTs.

DHTs, such as Chord, CAN, Pastry and Tapestry, are the cornerstone for any peer-to-peer storage system [12, 11, 54, 57]. At its core, Oceanstore uses a DHT, and builds

TABLE 2.1: *Features of DHTs:* The second column indicates how long a routing path, in terms of application layer hops, will be on average while the third column indicates whether the DHT can be built so that it can match the underlying physical topology of the network. N is the number of nodes in the DHT and d is the dimension chosen for CAN.

DHT	Route length in hops	Topologically sensitive
Chord	$O(\lg N)$	
CAN	$O(N^{1/d})$	✓
Pastry	$O(\lg N)$	✓
Tapestry	$O(\lg N)$	✓

global-scale storage services around it to provide consistency, reliability, and security [58]. Other services have also been built on top of DHTs, such as multicast (SCRIBE on top of Pastry, for example [14]), which use the DHT primarily for routing.

While peer-to-peer networks provide the kind of mapping we need for the long-term storage required by peer-to-peer multiplayer games, they are not optimized for providing low-latency interactions between millions of players in a virtual world. In the same respect, multicast is optimized to send as few messages as possible. While native multicast has not been widely deployed, application layer multicast provides a viable alternative to group communication. Unfortunately, current application-layer multicast protocols do not address the problem of malicious nodes in the system (a distinct possibility with games) and the additional latency introduced by the protocols make them insufficient as the sole networking solution for large-scale multiplayer games. Our work addresses these problems.

2.4 Multiplayer Game Research

Diot, Gautier and Kurose described the first protocol for distributed games [15, 59] and built a game called MiMaze to demonstrate its feasibility. Their work is important because they developed a technique called *bucket synchronization*, in which game time is divided into 'buckets', in order to maintain state consistency among players. The MiMaze protocol uses multicast to exchange packets between players, resulting in a low latency; however, it does not address cheating.

At the other end of the spectrum, Baughman and Levine designed the *Asynchronous Synchronization* (AS) protocol to address the problem of protocol level cheats [3]. As a building block, the AS protocol uses a protocol called the *Lockstep* protocol. Lockstep divides time into rounds. In one round, a player reliably sends a cryptographically secure hash of their move to all other players. Once all players have sent their hashes, they send their plain-text updates in the following round. This forces everyone to *commit* their move, without revealing it, thereby preventing anyone from knowing someone else's move ahead of time.

To mitigate the problem of delay introduced by reliable transport and to scale to a larger number of players than Lockstep can handle, Baughman and Levine developed *Asynchronous Synchronization* (AS). In AS, a player can advance forward in time asynchronously from other players, but must enter into Lockstep periodically with the other players. To determine when two players must use Lockstep, a sphere of influence is associated with each player. When a player receives or misses an update from another player during a round, the associated sphere is contracted or dilated respectively. This allows players to progress in rounds asynchronously until their sphere intersects with another player's sphere—at which point they must engage in Lockstep and wait for each other's messages.

AS is a major advance in distributed protocols because it is provably secure against the fixed-delay and timestamp cheats. It gains this security by forcing moves to occur in lockstep—no player can receive a plain-text move before they commit their move. Unfortunately, its main drawback is that its *playout latency*, which is the time from when an update is sent out to when the update can be displayed to other players, has a minimum bound of three times the latency of the slowest link. This delay is due to Lockstep's use of reliable transport. Because interactive games require latencies of 100-250ms, the minimum delay bound of Lockstep makes it unusable for games unless all players have fast links to each other.

Cronin et al. designed the *Sliding Pipeline* (SP) protocol [60] in order to improve the Lockstep protocol. They add an adaptive pipeline that allows players to send out several moves in advance without waiting for ACKs from the other players, reducing the time that is dead-reckoned between rounds. The pipeline depth is designed to grow with the maximum latency between players so that *jitter*, or inter-packet arrival time, is reduced.

Like the Lockstep protocol, the playout latency of the SP protocol is $3d$, where d is the maximum delay between any two players.

Chambers et al. developed a technique to mitigate the amount of information exposure in RTS games [61]. In their technique, a player's move is only revealed to another player if she is within the other player's viewable area. Otherwise, she sends a cryptographically secure hash of a secret key and her move. At the end of the game, each player reveals their secret keys and moves. The moves can be verified as being valid moves in the game by re-running the simulation.

Bharambe et al. design Mercury, a distributed publish-subscribe peer-to-peer communication architecture [4, 62]. Mercury is a type of DHT where users data is published and subscribed to according to its subject. The subject is hashed to generate a key which in turn can be routed to a node in the DHT. This node acts as the rendezvous point (RP) to gather and disseminate publications on the subject.

Mercury uses a subscription language that is a subset of relational database query languages. This allows users to subscribe to a range of subjects with a single query. For games, different areas in the virtual world are 'subjects' that players subscribe to so that they can send and receive updates to other players interested in the same area of the virtual world. Mercury has been shown to solve the distributed state maintenance problem for MMOGs [4].

Knutsson et al. also designed a publish/subscribe system [6] using Pastry [54] and Scribe [14]. The virtual world is statically divided into regions, and players in each region form a group. Each region maps to a multicast group through Scribe so that updates from the players are multicast to the group. Consistency is achieved through the use of *coordinators*. Every object in the game is assigned to a coordinator; therefore, any updates to an object must be sent to the coordinator who resolves any consistency problems. Fault tolerance is achieved through replication. However, their system does not fully take cheating into consideration in its design, but instead relegates it to future work.

In the federated peer-to-peer network game architecture, groups of players use a peer-to-peer protocol for communication which then send updates to a set of servers that are installed on the Internet [63]. The servers coordinate the migrations of players between servers and unicasts updates to relevant servers and players. Thus, the servers provide a

publish/subscribe system for players and an application layer multicast service, though at the high cost of locating multiple servers across the Internet.

The federated peer-to-peer game architecture acts simply as a client/server architecture, except that the typical cluster of servers are distributed on the network to be *closer* to players. Given the cost of locating and maintaining servers in geographically different areas, the effectiveness of this architecture is not clear. Certainly, simply maintaining a cluster of servers at a single location is more cost effective and will achieve higher performance (since a cluster will not suffer from poor network performance that is typical of the Internet). Further, the authors do not address cheating in their work.

In the game industry, very few networked games are fully distributed. One notable exception is Age of Empires (AoE) [9], in which games are synchronized across clients and peer-to-peer communication is used. AoE's protocol is similar to bucket synchronization, except that unicast is used. While AoE is a commercial success for distributed game protocols, it is subject to all but the inconsistency cheat (because players periodically exchange hashes of the game state with other players to detect inconsistencies).

2.5 A Taxonomy of Cheats

Cheating is defined as any action that circumvents the normal course of the application to the benefit of a user. Cheats are possible because of security flaws in the application, protocols, or network. We can taxonomize cheating into categories based on the layer which they occur at: network, application, or game. Table 2.2 lists several common types of cheating.

Cheats occurring in the first category, the network level, allow players to gain an advantage in the game by exploiting security flaws in network and routing protocols. Cheats occurring at the application level occur from applications modified from their original intent². Last, cheats occurring at the game level are cheats that occur in the game by breaking game rules (possibly by exploiting bugs or sidestepping rules in some way).

²Typically network and application level cheats both occur through modifying the application, though network cheats specifically target security flaws with the network protocols.

TABLE 2.2: *A taxonomy of cheating:* Check marks indicate whether this type of cheat is possible under the listed architecture. Stars indicate cheats which are partially possible.

Cheat	Level	Distributed		Client/Server
		P2P	Multicast	
Denial of Service	Network	✓	✓	*
Fixed Delay	Protocol	✓	✓	*
Timestamp	Protocol	✓	✓	*
Suppressed Update	Protocol	✓		
Inconsistency	Protocol	✓	✓	
Collusion	Protocol & Application	✓	✓	✓
Secret revealing	Application	✓	✓	*
Bots/reflex enhancers	Application	✓	✓	✓
Breaking game rules	Game	✓	✓	✓

In addition to showing the taxonomy of cheating methods, Table 2.2 shows which cheats can be used with the three primary architectures (peer-to-peer, multicast, and client/server). Both peer-to-peer and multicast architectures in this case are considered to be fully distributed architectures without a centralized server. We also do not consider a multicast client/server architecture because the cheats associated with the client/server architecture are not dependent on a unicast or multicast communication paradigm.

In the following discussion of the categories of cheats, we use three characters: Alice and Bob, two non-cheating players, and Mallory, the cheating player.

Network level cheats

Network level cheats are those cheats which occur at the network level and which are preventable only by security measures that solve these problems on a more general level. For example, the main network level cheat we list is a Denial of Service attack. In this cheat, Alice and Mallory are competing. Mallory sends a flood of bogus packets to Alice to cause her to lose important game related packets. Thus, Alice's game play suffers, giving Mallory an unfair advantage.

Clearly, two primary solutions exist to solve network level cheats: security measures and indirection. Security measures can be implemented at the network level that prevent denial of service (DoS) attacks. However, DoS attacks directed at a game may require less

traffic than those aimed at causing other applications to fail, such as a web server. This is because RIM applications typically have stringent latency requirements. Thus, even a few extra packets sent by Mallory to Alice may be enough to cause packet loss that gives Mallory an advantage.

Indirection, where Mallory has to forward updates to Alice through a relay which then forwards the updates to Alice, can also be used to solve DoS attacks on RIM applications. This helps solve the DoS problem because Mallory no longer knows Alice's IP address to perform the DoS attack on. However, this solution introduces additional delay. Note though that the server in the client/server architecture acts as this relay between clients, and hence a DoS attack on the client/server architecture will only affect the server.

Protocol level cheats

Protocol level cheats are cheats that occur by manipulating packets, or the flow of packets, from player to player. Distinguishing between a protocol or application level cheat is somewhat blurry because to manipulate the contents of a packet one often has to alter the application.

Many of these cheats can be accomplished by using something like a NIST box, which is a router that can be used to introduce artificial delay and packet loss on the network [64]. By placing the NIST box between the game and the network, a player can easily add delay or suppress updates from the game that is indistinguishable from real network conditions. Monitoring programs normally used to detect application level cheats also cannot detect these kinds of cheats. Furthermore, because these cheats are easy to implement and undetectable, they enable the less sophisticated game player to cheat without being caught.

- *Suppressed Update:* In this cheat, Mallory suppresses updates to Alice and Bob. Just before the game would disconnect him due to packet loss, Mallory sends a new update to his opponents. As a result, Alice and Bob do not know where Mallory is exactly or what actions he has performed, giving him the ability to 'hide'.

This cheat is particularly powerful when a player's update includes the actual location of a player instead of the series of moves (which is often the case when using unreliable protocols since a move might have been dropped). In this case, Mallory

does not let other players know where he is and then picks an advantageous location several rounds later.

- *Fixed Delay:* In this cheat, Mallory purposely adds a fixed amount of delay to his outgoing packets while accepting all incoming packets. This cheat allows him to receive updates faster than he is sending them, granting him the ability to respond to game events quicker. For example, Mallory has a 10ms connection to Alice, but artificially adds a 140ms delay on outgoing updates. Thus, Mallory can react to updates from Alice 10ms later, while Alice will not see Mallory's updates until 140ms after they occurred. The effect of disproportionate latency was examined in [65], which showed that the added latency directly affects the ability to win in some games.

This cheat is possible in the client/server architecture if the server allows players to timestamp their own updates (which may be done for performance reasons or to prevent clients from having to rescind moves they have already displayed on the player's screen). Typically, though, the server determines when events occur and local clients assume their updates were accepted by the server unless they are told otherwise.

- *Inconsistency cheat:* In the inconsistency cheat, Mallory sends his 'real' update to every player, except Alice, while sending a different update to Alice at time t . Now Alice thinks Mallory is in a different location than he really is, but every other player will disagree with Alice on Mallory's location. Later, Mallory can send updates to Alice that merge the two differing opinions on his location in order to hide his cheat. In the worse case scenario, Mallory can corrupt an entire game, but Mallory can also corrupt a single player, eliminating them from the game. The inconsistency cheat arises from the Byzantine General's Agreement problem [7]; in this case we are trying to agree on everyone's game state.
- *Timestamp:* Because events must be ordered for consistency purposes, a global clock is often used for time stamping. In the timestamp cheat, Mallory waits to receive an update from Alice and then sends his update with a timestamp that is *before* Alice's. For example, Mallory could send out a move with a timestamp earlier than the 'Alice

shoots Mallory' update just received. To other players, Mallory's message appears to have been delayed in the network and the shot misses.

The client/server architecture sidesteps this issue because the server can provide a total ordering on events (based on when it receives the update, not when the update is sent) and tells each client when each event occurred.

- *Collusion Cheat*: A collusion cheat occurs by having several players collude and either share packets or modify them in some way to gain an advantage over other players. For example, Mallory is colluding with Eve and is trying to catch Alice. Eve sees Alice, even though Mallory cannot, so Eve can simply inform Mallory of Alice's location. Recall that this occurs at the protocol level—in other words, Eve can simply forward Alice's positional updates to Mallory even though he shouldn't receive them.

Application level cheats

Currently, application level cheats are addressed by monitoring software installed on the computer that a game runs on. The monitoring software inspects computer memory, processes, and applications on the computer to detect if known cheats have been installed on the system. For example, the program can compare a CRC of important game files with known correct values to detect if the game has been tampered with. It can also scan processes to determine if rogue processes are running. Of course, like a virus scanner, this requires the monitoring program to have up to date information. Furthermore, one could theoretically run the game and the operating system in a *sandbox*, or emulated system, so that the monitoring system cannot detect cheating applications. While this technique may not always be practical due to performance issues, it can subvert cheat-detection software.

- *Secret revealing*: Secret revealing occurs by Mallory altering his game client to give him information that would normally not be available. For example, he may modify his client so that walls are translucent, giving him the ability to locate enemy players in a maze easily.

In the client/server architecture, this cheat can only be prevented by revealing secrets at the last possible moment. In the example above, the server would not reveal Alice's position to Mallory until Alice was in Mallory's direct line of sight. Unfortunately, this leads to high latency [66].

- **Bots/reflex enhancers:** This cheat occurs by modifying the client with additional software so that a player can react faster than humanly possible. For example, Mallory can automatically aim his weapons at Alice by reading Alice's position from the game client and firing at Alice's predicted location.

This kind of cheat is difficult to detect in a game. An extremely skilled player may look like a bot if her accuracy is very high.

Game level cheats

Game level cheats occur by finding loopholes in the game rules. For example, if the inventor of Poker had left out the rule that a player is not allowed to peek at an opponent's hand, then she could base her next move on her opponent's cards. Unfortunately, preventing loopholes in game rules is a non-trivial problem!

CHAPTER III

THE NEW-EVENT ORDERING PROTOCOL

3.1 Introduction

Our proposed solution for solving cheat-proof and real-time event ordering for distributed MMOGs consists of two parts: a real-time and cheat-proof event ordering protocol and a scalable, hierarchical structure for organizing a large number of peers so that messages are disseminated quickly while event ordering is maintained. In this chapter we present the New-Event Ordering (NEO) protocol which uses a majority voting system to both prevent cheating and ensure consistency in a timely manner. To achieve these important goals, NEO is limited in scalability. In Chapter 4.4, we show how to scale NEO.

The goal of any game protocol is to be *playable*, which we define as having three facets: consistency, resistance to protocol level cheats, and the timely delivery of updates. We address these three significant problems with the design of NEO. First, we have to ensure consistency of the shared state between players so that all players view the same sequence of events in the game. To that end, we have defined a new consistency model called *majority consistency*, and we show that NEO achieves majority consistency. With this new model of consistency, all players will experience the same sequence of events if those events were seen by a majority of players. This consistency model helps reduce the need to recover late or lost updates while ensuring that all players remain consistent.

Second, NEO has to be resistant to protocol level cheats. Any protocol that will be used for games must be resistant to protocol level cheats for the simple reason that games have a long history of players cheating through various means. We show how NEO can

prevent the four common protocol- level cheats under a broader definition of cheating than has been previously used.

Third, we have to deliver updates to players in a timely manner. Table 1.1 lists desirable playout latencies for different game archetypes, indicating how fast updates must arrive at their destinations. NEO divides time into *rounds* and uses the round length to bound the maximum latency of a player *from a majority of other players in the game*. This means that it is acceptable for a player to have a high latency to some players as long as her latency to most players is low. Assuming reasonable network conditions, NEO provides a playout latency 1/3 shorter than previous cheat-proof protocols, making it usable for all game archetypes.

NEO is the first protocol designed to simultaneously address consistency, security and low playout latency. Diot and Gautier’s Bucket Synchronization only considered the problem of synchronization and consistency [15]. Baughman and Levine’s Asynchronous Synchronization protocol addressed some protocol cheats, but was subject to inconsistencies [3]. Age of Empires managed consistency but was subject to cheating [67]. Table 3.1 summarizes the primary game protocols and their contribution in terms of a low playout latency, cheat- proofness, and consistency.

We present work directly related to NEO in Section 3.2 and the formal description of NEO in Section 3.3. We also show that NEO achieves majority consistency and prove that it is cheat proof in Section 3.3. We then provide a simulation study to show that NEO

TABLE 3.1: *Contributions of game protocols:* NEO provides a low playout latency, is cheat-proof and provides consistency. Bucket synchronization provides a low playout latency, but is not cheat-proof, and doesn’t fully provide consistency. AS/Lockstep provides a consistency, but is only partially cheat-proof and cannot ensure a low playout latency. Age of Empires provides a low playout latency but it is only partially consistent because it requires periodic exchanges of hashes of the entire game state between all players, ending the game if any inconsistencies were found.

Protocol	Low Playout Latency	Cheat-Proof	Consistency
NEO	✓	✓	✓
Bucket Synchronization	✓		<i>partial</i>
AS/Lockstep		<i>partial</i>	✓
Age of Empires	<i>partial</i>		<i>partial</i>

outperforms other cheat-proof protocols, that NEO maintains a low playout latency even in the face of late or dropped packets, and that group sizes do not affect the performance of NEO in Section 3.5. Last, we show how scalable NEO is considering by analyzing NEO group sizes versus payload sizes and available network bandwidth. We conclude in Section 3.6.

3.2 Background and Related Work

In order to understand concepts related to NEO, we present definitions, background material, and past work related specifically to NEO.

3.2.1 Terminology

The following terms are common to the game industry and used in NEO:

- *Update*: A message from the player which changes the state of the game. Interchangeable with *move* or *event*.
- *Round*: A fixed length amount of time. Rounds can be numbered sequentially from 0 to n starting from the beginning of the game. The time that a round begins at can be calculated by $round_length * round_number$.
- *Playout latency*: The time from when an update is generated to when the game can display the update to the player. Table 1.1 lists acceptable playout latencies for a variety of game archetypes.
- *Rollback*: Inserting a late update into its correct sequence in time and replaying all updates instantaneously up to the current time. This assumes that we have kept all updates in the past up to the point in time that the late updates occurs. Rollback also assumes that it is possible to replay all updates that happened previously, which is possible in the game world. For example, if updates u_1, u_3 were received and at a later time u_2 is received, the system will insert u_2 between u_1 and u_3 and replay u_2 and u_3 to determine the correct results of all updates.

- *Dead-reckoning*: Predicting the future state of an object in a game based on its past behavior. For example, we can *dead-reckon* that an object traveling on vector V at velocity $|V|$ will continue to do so in the future. To reconcile incorrect predictions, we use rollback when the correct update arrives.

3.2.2 Game Protocols

Several networking protocols for games have been created in either a research or commercial setting. We review these protocols which were originally discussed in Chapter II, but we discuss them in relation to which properties they satisfy: low playout latency, resilience to protocol cheats and consistency. We summarize these properties and how they relate to protocols in Table 3.1.

Low Playout Latency

Of the prior game protocols, *bucket synchronization* from Diot et al. and Age of Empires (AoE) provide a low playout latency for games [15, 59, 9]. In bucket synchronization, multicast is used to disseminate updates and players send updates at a fixed rate. Lost updates are not recovered and therefore the protocol can commit updates shortly after they arrive, leading to a low playout latency. The protocol used by Age of Empires is similar to bucket synchronization except that unicast is used. Thus, AoE also has a low playout latency. Neither of these protocols address cheating, though both partially address consistency. Our protocol NEO also has a low playout latency.

Resilience to Protocol Cheats

Asynchronous Synchronization (AS) was the first protocols to address protocol level cheats [3]. AS uses Lockstep which divides game time into rounds and progresses by first sending a hash of an update to all players over a reliable channel followed by sending the plain-text update to all players to reveal the moves. This allows players to commit moves and agree on the set of moves in one round without knowing the contents of those moves, thereby preventing the Timestamp cheat. However, Lockstep is subject to other

types of protocol cheats such as the Suppressed Update cheat, the Fixed-Delay cheat, and the Inconsistency cheat. NEO addresses all of the protocol level cheats.

Synchronization and Consistency

Bucket synchronization, Age of Empires, and Asynchronous Synchronization all address synchronization and consistency to some extent. With bucket synchronization and Age of Empires, updates are timestamped and committed at some point in the future after they are received. For example, if an update has the timestamp t , it is committed at time $t + d$, where d is a fixed amount of time set by the game. This acts as a synchronization point for all clients—ideally d masks various delays experienced by updates as they travel through the network and thus all players are committing moves at the same time. However, in both bucket synchronization and Age of Empires, clients can become inconsistent. Using the inconsistency cheat, a player can easily make a game using either protocol become inconsistent.

The AS protocol with Lockstep also partially maintains consistency. Players remain consistent because Lockstep uses reliable transport to ensure that all updates sent by a player are received by the other players. This leads to a consistent view of the shared state in the game as long as the player does not execute the Inconsistency cheat. NEO, on the other hand, can prevent the inconsistency cheat and we describe how it achieves consistency in Section 3.3.2.

3.2.3 Consistency and Event Ordering

Consistency models were originally created to address the problem of correct program execution when multiprocessor systems share one or more memory units. Because this shared state can be written to or read from more than one processor at a time, thereby creating a condition where two processors disagree about what the contents of a memory location should be, the shared state must be updated in a predictable manner. Using a consistency model helps the programmer prevent inconsistencies and ultimately bugs. A wealth of research in computer architecture and distributed systems addressed the problem of (memory) consistency for systems ranging from shared-memory with local caches to dis-

tributed systems based only on message passing [28]. The following notions of consistency are relevant to the model we introduce for NEO.

Definitions:

- **Atomic (strict) Consistency:** The most strict consistency model where the result of any computation is the same as if all reads and writes follow a sequential order, and the reads and writes of each individual processor appear in this sequence in the order they actually occurred at [68].
- **Sequential Consistency:** The result of any computation is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program [8]. [69].

With *atomic consistency*, all reads and writes have a sequential order and this order follows the *real-time* order that each read and write have when they are executed on the individual processors. However, atomic consistency is essentially impossible to achieve in a true distributed system due to the simple fact that we can only synchronize clocks to within some δ and therefore we can never know the actual time when a read or write occurs [70].

In Lamport's *sequential consistency*, all reads and writes follow a sequential order based on the program order of reads and writes running at each processor. Thus, a programmer can assume that reads and writes will never be reordered in the execution of the program on multiple processors.

In a client/server architecture for games, the client acts as a cache of data it receives from the server. When the client wishes to change the state of the game, it sends a message to the server in the form of an update. The server validates the update and responds to all clients with the change to the state they are caching for the players. Thus, the server determines the order of all reads and writes to the state of the game.

In a distributed architecture, players share the state of the game and must agree on changes to that state. Like a multiprocessor system, each node acts as a cache and proces-

sor. Changes to the shared state must be propagated to the system so that all players maintain consistent views of the shared state. In Section 3.3, we introduce a new consistency model, called *majority game consistency* for games which leads to improved performance over traditional sequential consistency.

3.2.4 Time Warp and Rollback

Jefferson's Time Warp algorithm is a consistency algorithm designed specifically for distributed systems that do not require locking or other synchronization methods. Jefferson described *virtual time* as a more natural ordering of events in a distributed system than sequential event ordering. The concept of virtual time is conceptually like Lamport's regular *clock condition* [70], except that local events do not always advance the time¹. Using virtual time, Jefferson defines the Time Warp algorithm that is used to synchronize distributed systems.

Time Warp works as follows. When a message is sent, the message is timestamped according to the current local virtual time. When a message is received, virtual time is advanced to the timestamp contained in the message if it is greater than the local time.

Time Warp gets its name because a system can rollback all events that have occurred whenever it receives a message from another system with a time earlier than its current virtual time. Thus, a node in the system can execute arbitrarily far into the future, including sending out events to other systems, but must rollback time whenever it receives an event from an earlier time period.

Jefferson also provides an algorithm for estimating the global virtual time (GVT), which is the earliest virtual time of any process currently executing in the system, and proved that the GVT increases monotonically. The GVT can be used to determine how long old events must be tracked because a system will *never* be rolled back to a time earlier than the GVT. The GVT can also be used to determine when to commit to I/O or when a global snapshot of the system has completed.

For NEO, we develop a similar notion of a moving time horizon behind which no rollback can occur.

¹In other words, the 1st condition of the *happened before* relation is not used to define virtual time.

3.3 The New-Event Ordering (NEO) Protocol

The New-Event Ordering (NEO) protocol is the first protocol that provides consistency, avoids the common protocol level cheats, and keeps the playout latency to a majority of players within a given bound. NEO divides time into rounds and derives its name from the fact that new events are ordered according to which round they arrive in.

In all of our discussions about NEO, we assume a typical model for the network and players. First, we assume that players only communicate by message passing over a best-effort network. Thus, packets may be delayed, lost, or arrive out of order. We assume that messages are transported by a protocol such as UDP and therefore arrive uncorrupted². Initially we assume that a group of players that is using NEO will continue to do so until the game is over. We later relax this assumption to allow the joining and leaving of players in a NEO group. We assume that players can modify NEO packets, artificially delay or drop packets, though they cannot collude. We discuss collusion prevention in Section 3.3.3.

3.3.1 The Basic NEO Protocol

At a high level, NEO provides consistency and prevents cheats through two main techniques:

- Players commit an update by first sending the digital signature of the update. Players later reveal the plain-text update which allows other players to verify that the digital signature matches the plain-text update.
- Only moves which are received by a majority of players within a given time bound are accepted in the overall ordering of events. If a player is in the minority of those who missed an accepted update, NEO continues, but the player must request the missing update and rollback once the update is recovered.

²Recall that UDP uses a checksum to determine if a packet is has been corrupted en route to its destination. If the checksum is invalid, the packet is dropped and is never delivered to the operating system.

NEO Rounds

With NEO, time is broken into equal intervals, called rounds, with a time length of R . At the start of the game, all clocks are synchronized to within some δ , where $\delta \ll R$, using a mechanism such as the Network Time Protocol (NTP) [27]. Rounds increase sequentially, starting from round 0 at the beginning of the game.

At the start of each round, NEO generates an update to the state of the game based on the player's input and sends out a digital signature of this update to all other players. At the start of the following round, NEO sends out the plain-text update. When a player receives the plain-text update, she can use the digital signature to recover the hash of the update and verify that the hash matches the update. Each message M from player A at round r has the following format:

$$M_A^r = S_A(U_A^r), U_A^{r-1} \quad (3.1)$$

In this message, $S_A(U_A^r)$ is the digital signature³ for the update for round r . U_A^{r-1} is the plain-text update for the previous round. Figure 3.1 shows how NEO sends digital signatures of updates followed by the plain-text updates the following round.

NEO rounds are important to both the performance and security of the protocol. Rounds have the following characteristics:

1. Every player first commits a move in one round by sending their digital signature of the move. The signature acts as proof of the player's move for that round and prevents her from using a different move when it is revealed the following round.
2. As with Lamport's logical clocks [70], events that arrive in the same round, occur at the same time, but we order the commitment of these events by a mutually agreed ordering mechanism. For example, all accepted updates for round r occur at time t in the game, where $t = r * R$ in the game. The mechanism for deciding the actual order to commit all the updates for round r could be any number of techniques, such as a

³The digital signature S is calculated by $S = E(H(M), private_key)$, where we encrypt the hash of the message, $H(M)$ by the user's private key. The hash can then be recovered by using the public key of the user and compared with the plain-text message that will later be revealed to verify that the message did indeed originate from the user.

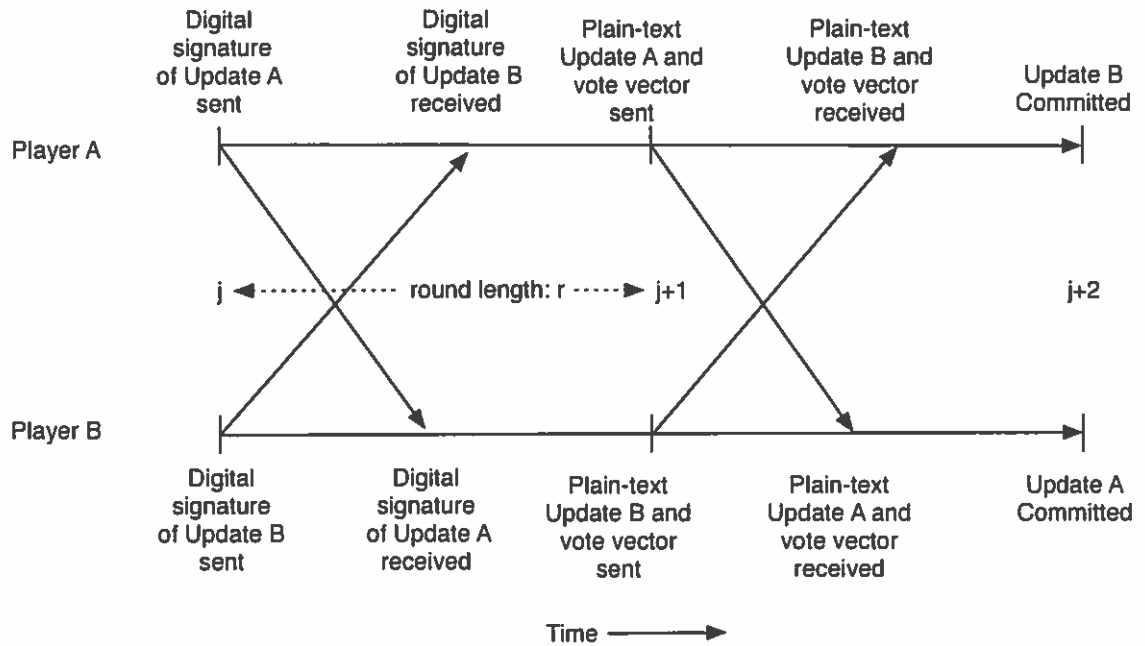


FIGURE 3.1: *NEO rounds:* Players first send the digital signature of their update for round r , followed by the plain-text update in round $r + 1$. By $r + 2$, NEO commits the update from round r . Thus, the payout latency of NEO is $2R$, where R is the round duration.

random number generated with t as the seed. The main stipulation on a tie-breaking technique is that it cannot be exploited by a player to force an ordering of the updates.

3. Round lengths serve as a bound on the delay that an update can incur before being received by other players. If the delivery time of an update exceeds the round length, the update is considered invalid. NEO's mechanism for handling invalid updates is described later. Bounded latencies ensure that the game progresses, unlike other protocols which use reliable transport and may experience unnecessarily long delays waiting for updates to be recovered.
4. All event ordering in NEO is based on round numbers, rather than timestamps. Updates are generated at round r and the plain-text updates are committed at the end of round $r + 1$ according to a mutually agreed upon order. Note that while the game time of the update for round r can be calculated by $round_number * R$, using rounds instead of timestamps prevents players from increasing or decreasing the time that an update occurred at by ϵ , thereby preventing the timestamp cheat.

5. NEO rounds lengths are tuned for the type of game being played. Games which require a shorter playout latency use a shorter round length. For example, the desirable playout latency of a role-playing game is approximately 250ms. Using a round length of 125ms ensures that all updates are revealed at 250ms.

Majority Voting

NEO has a distributed voting mechanism in the protocol to maintain consistency between players and to handle with late updates. In addition to sending the digital signature and plain-text update from the previous round, we add a voting vector representing updates that were received the previous round. The NEO update M from player A at round r with the voting vector V has the following format:

$$M_A^r = S_A(U_A^r), U_A^{r-1}, (V_A^{r-1} + S_A(V_A^{r-1})) \quad (3.2)$$

In this message, $S_A(U_A^r)$ and U_A^{r-1} are identical to the message in 3.1. $V_A^{r-1} + S_A(V_A^{r-1})$ is the digitally signed vote vector from player A for the previous round (see Figure 3.2).



FIGURE 3.2: *NEO update with voting:* The format of a NEO update includes the digital signature of the update for round j , the update for round $j-1$, and the vote vector for round $j-1$. Note that the update includes information such as the round number so that NEO can easily determine which update belongs to which round.

To understand how voting works, assume five players are in a game, and player A is tallying the votes from the previous round. Also assume that the majority⁴ is $\lfloor n/2 \rfloor + 1$. Table 3.2 lists the voting bit-vectors that each player has sent to player A . From the tally, we can conclude that a majority received A , B and D 's updates, while a majority did not receive E 's update (so it is considered invalid). As for player C , player A cannot determine what the outcome of the vote is, so she must contact another player to determine the outcome.

⁴One could set the 'majority' to be higher than $\lfloor n/2 \rfloor + 1$, but not lower since this could lead to an inconsistent state of the game. Theoretically, one could set it to higher than 50% to increase security at the cost of having to ignore more packets.

TABLE 3.2: *Vote table for a NEO player:* Player A receives a voting vector from all other players and the calculates which updates to accept and which to reject based on a majority. In this case, any tally of 3 or more positive or negative votes indicates an accept or reject respectively.

Player	Bit-vector
A	1 1 0 1 0
B	0 1 0 1 0
C	1 1 1 0 0
D	1 1 1 1 1
E	<i>packet lost</i>
Voting tally	3 4 2 3 1

Once all votes are collected from all other players, NEO tallies the votes with four possible outcomes:

1. *Commit:* If NEO receives the update, along with a majority of accept votes, the update is committed.
2. *Reject:* If NEO receives a majority of reject votes, the update is rejected.
3. *Recover votes:* If NEO receives the update, but does not have a majority of accept or reject votes, NEO must recover a sufficient number of votes.
4. *Recover update:* If NEO did not receive the update, but has received a majority of accept votes, NEO must recover the update.

The process of recovering missing votes or updates may cause NEO to rollback to ensure consistency.

Voting serves two primary purposes:

1. Voting allows only those updates received by a majority of players within the round to be required for consistency. This allows NEO to quickly reject updates that are late or missing. Because the updates have been received by a majority of players, only a minority of players will be required to recover an update if they missed it. Assuming a majority of players are connected and receiving updates on time, NEO will continue to progress through rounds⁵. On the other hand, if we used reliable

⁵If a majority of players are not receiving updates from each other, then the game is not playable, but this holds true for any game, distributed or not!

transport, dropped or late packets would delay the game for all players until it was recovered.

2. Voting prevents players from cheating by purposely delaying or dropping updates. The most that a player can delay an update to any player is by the difference of the round duration and the network latency to another player. Beyond that, the update will be considered late. However, since updates are not committed until the end of the round, this delay has no effect on the game.

With the majority voting mechanism, all players will eventually agree on which updates have been accepted and which have been rejected, assuming that a majority of players can communicate with each other.

As with Jefferson's virtual time [71], NEO has a similar concept to global virtual time (GVT), which we call the *NEO event horizon*. Recall that the GVT is the earliest point in time that the system may need to rollback to. Jefferson proved that the GVT advances monotonically with the Time Warp algorithm [71]. We define the NEO event horizon as follows:

Definition 3.3.1. NEO Event Horizon: The earliest round that any player may need to rollback to due to the recovery of missing updates or votes, i.e. the NEO event horizon is a minimum round r of all players such that if the current round is k , $r \leq k - 2$ and NEO will never rollback before r . Later we will show that the NEO event horizon advances monotonically over time.

The NEO event horizon is important for two reasons. First, by estimating how far back the event horizon is from the current round, we know the earliest time that NEO would possibly rollback to. Thus, state in the NEO group is stable prior to the event horizon. Second, players need to only keep state for rollback purposes back until the event horizon. Once the event horizon advances, they can discard the stored state.

Estimating how far back the NEO event horizon is from the current round is difficult primarily because we cannot predict network conditions. However, if we assume that we need to recover update u for round r , NEO will not detect this fact until the start of round $r + 2$. If NEO then sends out a request for the missing update or for the missing votes in

round $r + 2$, which will arrive at the latest by $r + 3$, then the other NEO peer will respond by round $r + 4$ with the needed information. However, NEO normally commits updates at $r + 2$, so while the NEO event horizon in the best case here would be 4 rounds prior to the current round, it is only two rounds behind the normal commit time for an update. We explore the effects of packet loss and update/vote recover in our performance analysis of NEO using simulation in Section 3.5.

3.3.2 NEO Consistency

The purpose of event ordering protocols for games is to maintain consistency for all players. We formally define three types of consistency. The first two follow traditional consistency models for distributed systems, while the third is a new consistency model we have developed.

It is important to understand that NEOs notion of consistency applies to updates, not to notions of consistent actions in the game world. In other words, NEOs protocol yields an ordered set $S(U)$ of updates that are mutually agreed upon by a majority of the players. When rollback occurs on player A in NEO, the late update is inserted to bring player A into agreement with the majority. Rollback does not change any events that have already been accepted by the majority. In the game world, rollback may cause unrealistic things to occur, e.g. a dead character suddenly is alive again. However, as long as all players have agreed to the same sequence of updates $S(U)$ and as long as for each player, her updates in $S(U)$ corresponds to the order in which she generated them, NEO is consistent.

Definitions:

The first two consistency models are based on the corresponding notions of traditional distributed systems described in Section 3.2:

Definition 3.3.2. Atomic (strict) Game Consistency: The result of any game is the same as if the updates from all players were executed in some sequential order, and this order follows the order in which updates were actually issued by players. This model is analogous to *atomic consistency* [68].

Definition 3.3.3. Sequential Game Consistency: The result of any game is the same as if the updates from all players were executed in some sequential order, and the updates of each individual player appear in this sequence in the order specified by the player. This model is analogous to Lamport's *sequential consistency* [8].

Note that strict consistency means there is one global sequential ordering of events while sequential consistency allows any interleaving of events that is consistent with local ordering.

Strict game consistency assumes that some centralized system can actually determine and sequentialize the order of updates received from all nodes in the system, that all nodes will see this ordering, and that within this order, nodes will see their updates in the same order that they were issued. For example, a client/server architecture exhibits strict game consistency because the server can timestamp each event as it arrives and prevent the re-ordering of events from individual clients.

With *sequential game consistency*, the protocols must ensure that however updates are interleaved in the global sequence of updates, each individual player sees their updates in the order that they were issued. For example, if the player issues update A before update B, then update A must occur before update B in the sequential order of all update in the system.

The third consistency model is a new model we developed for distributed games and which NEO provides:

Definition 3.3.4. Majority Game Consistency: The result of any game is the same as if the updates from all players were executed in some sequential order, and only the updates seen by a majority of players appear in this sequence in the order specified by the originating player.

In *majority game consistency*, all players see the same sequence of updates, however only those updates seen by a majority of players appear in this sequence. As long as all players can agree which updates have been seen by the majority of players, then each player can individually recreate the sequence of updates, leading to consistency between players. A key feature of majority game consistency is that updates not agreed on by a majority are

dropped, i.e. they are not considered in the set of ordered events. Thus, late updates are not accepted by NEO and do not cause rollback to occur. NEO provides majority game consistency through its distributed voting mechanism.

We now show that NEO can provide sequential game consistency if all updates are accepted each round. We then extend the proof to show that it provides majority game consistency when only some of the updates are accepted. We divide our proofs into two cases, with and without rollback. Note that NEO uses rollback only in two circumstances:

1. An update has arrived, but NEO does not have a sufficient number of votes to decide whether to accept or reject the update. NEO then requests the missing voting vectors and decides once it has a sufficient number of votes to accept or reject the vote. NEO will need to rollback updates to insert this update if the update was accepted.
2. An update did not arrive, or arrived late, and NEO has determined it should accept the vote. In this case, NEO will rollback updates and insert this missing update.

We first show that the NEO event horizon exists and that it increases monotonically over time.

Theorem 3.3.5. *At round $k \geq 2$, there exists a minimum round r , where $0 \leq r \leq k - 2$, for all players which NEO will not rollback behind.*

Proof. Assume the current round is k , the update we are rolling back to is u_r and the round that Player A is rolling back to is r . By definition, rollback is the removal of all previously committed events which occurred from $r \dots k - 2$ followed by the subsequent commitment of the events which occurred from $r \dots k - 2$, including u_r , at time k . Unlike the Timewarp algorithm [71], the insertion of u_r into the sequence of events from $r \dots k - 2$ does not cause the generation of any new events, even though the addition of u_r may cause future events to be invalid. Since no new events are introduced into the system, adding u_r to Player A's sequence of events does not cause any further rollback by Player A or by any other player in the game. Therefore, NEO will not rollback before r .

Note that the only reason Player A had to rollback was because it was in the minority of players that missed update u_r and discovered that u_r was accepted by a majority of players.

Players in the majority will have accepted u_r and will therefore not change their sequence of events from $r \dots k - 2$. Thus, majority players will also not cause NEO to rollback before r . \square

Lemma 3.3.6. *The NEO event horizon never decreases.*

Proof. By Theorem 3.3.5, we know r is the minimum round that NEO can rollback to. Thus, NEO cannot rollback to $r - k$, where $k > 0$, since then $r - k$ would be the minimum round. Therefore, the NEO event horizon never decreases. \square

Theorem 3.3.7. *The NEO event horizon monotonically increases over time.*

Proof. By Theorem 3.3.6, we know that the event horizon will never decrease. Therefore, we will show that it will increase. We assume that the network is not partitioned and that a majority of players can contact each other.

If the NEO event horizon is at round r , Player A needs to either recover the update u_r for round r , or a sufficient number of accept or reject votes for the update for round r . Assume Player A is in the majority of connected players. Then, there must exist a path from Player A to some other Player B with the update or the necessary votes to decide on u_r . Thus, Player A can recover the needed information and decide on u_r . Once all such players have done so, the NEO event horizon will advance to at least round $r + 1$.

Assume Player A is in the minority of players not connected to the majority of players. Player A's updates or votes will not affect the majority and thus the majority will continue to decide on updates and the NEO event horizon will advance.

Thus, the NEO event horizon will increase monotonically with time. \square

Theorem 3.3.8. *Prior to the NEO event horizon, if all updates from all players have been accepted, NEO is sequentially consistent.*

Proof. We divide our proof into two cases, without and with rollback. NEO uses a mutually agreed upon tie-breaking mechanism to order all events that occur in round j . We assume the current round is k .

- (i) *No rollback:* Without rollback, the NEO event horizon must be at round $k - 2$. Since we assume that all updates have been accepted from all players, then all updates will

appear in global sequence to all players. Rounds prevent a re-ordering of events, and thus the sequence of events generated by players will be interleaved in the global sequence of events. Thus, NEO is sequentially consistent at round $k - 2$.

- (ii) *With rollback:* With rollback, the NEO event horizon is at some round $r \leq k - 2$. Thus, all updates for rounds prior to r have been accepted by all players. All updates from all players appear as a global sequence consisting of an interleaving of the sequence of events generated by individual players. Thus, NEO is sequentially consistent at round $r < k - 2$.

□

Definition 3.3.9. We define a *partition* to be a subgroup of players in the game who are able to send and receive messages within a time bound of Δ . If $\Delta > r$, where r is the round length, this partition is unable to participate in NEO and removed from the game.

Definition 3.3.10. We define a *majority partition* to be a partition in the game which has m out of n total players in the game, where $m > \lfloor n/2 \rfloor + 1$.

We assume that a majority partition in the game exists. If a majority partition does not exist, NEO cannot achieve majority game consistency. However, we assume that players in a partition will eventually decide that players outside of the partition are no longer reachable, and remove them from the game. At that point, the partition will become a majority partition and the following proofs hold.

Theorem 3.3.11. *Prior to the NEO event horizon, NEO provides majority game consistency to a majority partition in the game.*

Proof. We divide our proof into 3 cases, the first without rollback, and the last two with rollback. Recall that in round k all players send out the digital signatures of their updates and in round $k + 1$ all players send out the plain-text hashes and the vote vectors representing which signatures were received in round k . Because we assume no collusion occurs, we assume that a majority of players do not lie about which updates were received. We also assume that we have n players and the majority m equals $\lfloor n/2 \rfloor + 1$. We assume the current round is k .

- (i) *No rollback*: Player A receives all the digital signatures in round $k - 2$ and all the plain-text updates and vote vectors in round $k - 1$. Majority game consistency is achieved for this player because she can tally the vote vectors received by each player and accept those updates receiving a majority of votes while rejecting those updates receiving a majority of negative votes.

If a player is only missing a minority of either the digital signatures or plain-text updates with vote vectors, then as long as the player still receives a majority of accept or reject votes for each update, then the player will still correctly decide which events are valid because all players receiving the majority will have decided on the same course of action.

- (ii) *With rollback, majority partition*: Player A does not have a majority of positive or negative votes but has received update u , or Player A has not received update u but has received a majority of accept votes for update u . In both situations, Player A must either recover enough votes from other players or must recover update u_r . Without a loss of generality, we assume that Player A is at the NEO event horizon at round r .

Assume that Player A is in a majority partition, she has to recover enough votes to decide on update u_r and that she has received p accept votes and q reject votes for the update. Then, Player A must recover either $m - p$ accept votes or $m - q$ reject votes. By definition of Player A being in a majority partition, she only needs to contact Player B in the majority partition who has already decided on update u_r (and by definition of the NEO protocol, can provide u and digital signature as proof of the validity of u).

In fact, if the majority partition has $o \geq m$ players in it, then any player that Player A contacts has a probability of m/o to have already decided on u . Once Player A contacts at most $o - m$ other players in the majority partition, she can decide on update u_r . By definition of the majority partition, Player A can indeed contact $o - m$ other players and therefore majority consistency can be reached for round r when all such players decide on u_r .

If Player A is not in a majority partition, her updates and votes cannot influence the

decision to accept or reject update u_r at round r . Thus, the majority of players in the majority partition will achieve majority game consistency without Player A for round r .

- (iii) *With rollback, non-majority partition:* Player A does not have a majority of positive or negative votes but has received update u , or Player A has not received update u and has received a majority of accept votes for update u . In this case, Player A is unable to reach any player in a majority partition, thus she cannot achieve majority consistency. Eventually, Player A will reconnect with the majority partition or her partition will remove the other players from their game and her partition will become a majority partition, at which point cases (i) and (ii) hold.

□

We now show the safety and liveness of NEO. The *safety* of NEO ensures that no error condition arises during the execution of the protocol, while *liveness* ensures that the protocol does not halt. The purpose of NEO is to maintain consistency, therefore we assume the error condition that may occur is that players in the majority partition become inconsistent.

Lemma 3.3.12. *The NEO protocol is safe; therefore, no error condition occurs for players in the majority partition.*

Proof. By Theorem 3.3.11, NEO provides majority game consistency to a majority partition in the game. Thus, NEO is safe. □

Lemma 3.3.13. *NEO is always live; thus, rounds advance monotonically with real time.*

Proof. By Theorem 3.3.7, the NEO event horizon advances monotonically with real time. Thus, NEO is live. □

Finally, we show that NEO provides majority game consistency at the end of the game. In other words, when the game ends, the NEO event horizon will eventually reach the end of the game and all players will be majority game consistent.

Lemma 3.3.14. *NEO provides majority game consistency at the end of the game.*

Proof. By Theorem 3.3.11, majority game consistency exists prior to the event horizon. By Theorem 3.3.7, the event horizon advances monotonically with time. By Lemma 3.3.13, we know NEO is live, therefore, when the game ends, the event horizon will eventually reach the final round of the game. Thus, NEO provides majority game consistency at the end of the game. \square

3.3.3 NEO Security

NEO is resistant to all of the protocol level cheats listed in Table 2.2, including the timestamp cheat, the suppressed-update cheat, the fixed-delay cheat, the inconsistency cheat, and the collusion cheat. Our holistic design of NEO which addresses the problem of real-time event ordering and cheating prevents the protocol level cheats through fixed-length rounds and majority voting.

Timestamp Cheat

The timestamp cheat is prevented by NEO through its use of rounds and cryptographically secure hashes. In the timestamp cheat, a player sets the timestamp of their next update to occur before the timestamp of an update they have just received. This allows the move to occur *before* the other move a player has just received. With NEO, players send a secure digital hash of a move before the plain-text move is revealed. Thus, by the time a player receives the plain-text update, it is too late to change the timestamp on the move they have already committed by sending the hash the previous round. Any changes in the move they had committed to previously will result in a different hash and therefore other players will discover that the player is cheating.

Suppressed-Update Cheat

The suppressed update cheat is prevented by NEO through its use of fixed-length rounds. In this cheat, players purposely drop packets (typically for several seconds) in the game to receive an unfair advantage over other players. In NEO, the round length determines the maximum amount of time any player will wait for a packet from another player. If a majority of players do not receive the suppressed update on time, then they will simply ignore the

move and assume that player did nothing for that round. In this case, the cheating player gains no advantage by suppressing updates.

In the case where one player suppresses updates to a minority of players, NEO is capable of recovering the missing updates within 2 round lengths. Since suppressed updates are most effective in games requiring a low-latency, we can assume the round length will be short to accommodate the game, and thus the missing updates will be recovered in time to make the cheat ineffective.

Fixed-Delay Cheat

In the fixed-delay cheat, players purposely add a fixed amount of delay to outgoing or incoming packets. If we assume that as long as packets which arrive within the round will not somehow give a player an unfair advantage over another player, then NEO prevents this cheat through the use of fixed-length rounds. The most a player would be able to delay incoming or outgoing packets would be d , where d is a delay such that $d + latency_to_player < round_length$. If the packet is delayed beyond that, the other players will vote that it was not received on time. On the other hand, if the packet arrives on time even with added delay, it will not give the player an unfair advantage since it can only be delayed less than one round length in duration.

We can assume that packets that arrive within a round do not give a player an advantage over another player because the use of fixed length rounds changes the concept of time in the game from being completely continuous to being discrete. As such, the only possible time that a move occurs at is at the start of a round, regardless of when the update arrived during the previous round. For example, assume that a round has duration d , begins at time t and ends at time $t + d$. Even though an update may arrive at $t + k$, where $t + k < t + d$, the move actually occurs at $t + d$, not $t + k$. Thus, the fixed-delay cheat is unable to give a player an advantage.

Inconsistency Cheat

In the inconsistency cheat, players send two different updates to two different players. For example player A sends update U to player B and update U' to player C. Note that this

problem is the distributed consensus problem. All non-faulty players should agree on the value proposed by A. In order to prevent this cheat, players need to exchange information regarding other players' moves.

We propose two different techniques to prevent the inconsistency cheat. In the first technique, players simply exchange the digital hashes of moves by other players instead of a vote that a move occurred. In essence, the bit-vector of votes is replaced by a list of hashes received the previous round. With this added information, players that send two different hashes for a round can be discovered. Indeed, in one round, at least one player will discover that the player has cheated and can report this information to the other players. The main drawback with this technique is the increased amount of overhead in each packet to send the hashes instead of the bit-vector.

In the second technique, players keep a log of all committed moves and hashes and they periodically perform a cryptographic hash of the state of the game and exchange those hashes. If the hashes do not match, then we know someone has attempted the inconsistency cheat and we can exchange logs of moves to determine who the culprit is. The frequency that we perform game state hashes depends on how quickly we want to catch potential cheaters and how much storage we want to allocate for committed moves. The technique of periodically hashing the game state has precedence in Age of Empires [67], though the game did not try to determine who the cheater was. The drawback of this technique is that depending on the amount of game state that needs to be hashed, the cost of hashing might be computationally expensive to perform frequently.

Collusion Cheat

In the collusion cheat, several players agree on some action that gives them an unfair advantage over the other players. The main collusion cheat that is possible in NEO is the modification of the voting bit-vectors. For example, a group of players lies and states that they did not receive a particular player's updates causing those updates to be rejected.

This cheat is only possible in NEO if a majority of players are colluding. However, few secure systems remain secure when a majority of participants are colluding in the system. In the unlikely event that a majority of players collude, NEO cannot prevent it.

On the other hand, out-of-band techniques can reduce this possibility. For example, we can prevent users from having more than one ID through authentication so that they cannot artificially create a majority on their own. With commercial games, IDs are tied to credit cards, thereby limiting the number of IDs any one person can have. Further, if collusion is a problem, we could require that packets be received by $\lfloor n/2 \rfloor + k$ of the n players instead of only $\lfloor n/2 \rfloor + 1$, where k is adjusted to make gaining a majority more difficult.

3.4 NEO Real-time Responsiveness

The primary goal of NEO is to be playable, which encompasses consistency, resistance to protocol level cheats, and real-time responsiveness. We define NEO real-time responsiveness as having three characteristics:

1. *Update Frequency:* A game protocol needs to be capable of sending and receiving updates each second at a frequency specific to the game archetype it was designed for under various network conditions.
2. *Playout Latency:* A game protocol needs to be capable of sending and receiving updates within a given latency specific to the game archetype it was designed for under various network conditions.
3. *Rollback:* A game protocol needs to keep rollback at a minimum since this effects a player's performance in the game.

These three metrics are vital for a game protocol to have real-time responsiveness. A protocol that does not attain the update frequency or playout latency specific to the game archetype it was designed for will not be playable to the participants of the game. A protocol which is subject to frequent or long rollbacks will hamper the players' ability to play the game since they will not be able to accurately predict the movement and behavior of other players. Table 3.3 lists the update frequency, playout latency and rollback percentages that NEO needs to achieve for real-time responsiveness.

The real-time responsiveness of NEO is affected by several variables, including the overhead of NEO, the ability to pipeline updates, and network conditions. We explore

the first two variables, overhead and pipelining, and then address network conditions in Section 3.5.

TABLE 3.3: *Playable parameters for games:* Playable update frequencies, playout latencies, and allowable rollback percentages for standard multiplayer game archetypes. While a higher update frequency, lower playout latency and zero rollback is always desired, these values represent reasonable values that protocols should target for real-time responsiveness from a player’s perspective.

Archetype	Update Frequency	Maximum Latency	Allowable Rollback
First-person/Racing	10-20 updates/s	100ms	3%
Role-playing Games	5-10 updates/s	250ms	5%
Real-time Strategy	5-10 updates/s	250–500ms	5%

3.4.1 NEO Overhead

The NEO protocol naturally has more overhead than simple distributed protocols or client/server protocols. The overhead in NEO is due to two main factors. First, NEO is resistant to the timestamp cheat. Like the Lockstep protocol, NEO first sends a hash of the update followed by the plain-text update [3]. In addition, NEO uses a majority voting system for consistency and cheat-prevention. This adds an additional amount of overhead to each update.

If we take a base protocol that simply unicasts updates to all players, we can calculate the amount of data that would need to be sent by each player. Assume that each player samples input and sends an update r times per second. Each update is unicast to the other $n - 1$ players. The payload of each update is p . The amount of data sent and received by each player is then $rp(n - 1) + rp(n - 1) = 2rp(n - 1) = O(n)$. Of course while this base protocol is $O(n)$, the hidden $2rp$ constant cannot be easily ignored. Without multicast, this is the minimum amount of data that will be sent with a peer-to-peer protocol and serves as a baseline for comparison with other peer-to-peer protocols.

NEO adds a digital signature and a digitally signed vote vector to the baseline traffic. The size of the digital signature depends on the size of the encrypted hash using public-key cryptography. A typical cryptographically secure hash, such as SHA-1 or SHA-256, requires 256 bits for the hash. Public-key cryptography encrypts messages into blocks

based on the size of the key. For example, RSA's algorithm encrypts messages into 128-byte blocks. We will simply refer to the size of the signature as k .

The vote vector increases in size depending on the number of players in a NEO group. Using a single bit per player, the vote vector adds $O(n)$ to each update. For explanatory purposes, we refer to the vector size as v . Thus, the total cost of NEO per second, where p is the payload size and r is the number of updates each second, is $r(k + p + v)(n - 1) + r(k + p + v)(n - 1) = 2r(k + p + v)(n - 1) = O(n^2)$. Again, we cannot easily ignore the $2r(k + p)$ hidden constant, but we can see that NEO adds an additional order n cost for secure, real-time event ordering for peer-to-peer games.

3.4.2 NEO with Pipelined Rounds

In the basic protocol, the delay from each player to the majority of other players is bounded by the duration of the round. Increasing the round length increases the length of time which the game must dead-reckon the positions and actions of other players. During this period of time, player input is ignored and the game may be inconsistent with the true state of other players. To address these problems, NEO can pipeline its rounds, similar to the technique of pipelining instructions in a processor and to the SP protocol [60]. The pipeline depth is related to the round duration and the update frequency, as seen in Figure 3.3. This relationship can be expressed in the following formula:

$$\text{pipeline depth} = \frac{\text{round duration}}{\text{update frequency}} \quad (3.3)$$

This formula has two important properties. First, the pipeline depth is 1 whenever the update frequency is equal to the inverse of the round duration. In essence, the basic NEO protocol has an update frequency of $1/R$, where R is the round duration. Second, if the pipeline depth is not an integer, then sending the vote vector and plain-text update at the start of each round will increase the ployment latency. Alternately, one could decouple the sending of the plain-text update and vote vector from the signatures by sending out the plain-text update and vote vectors after R has passed from the start of the round which the update belongs to.

We can now generalize Equation 3.1 using the pipeline depth d , round number r for player A in the following equation:

$$M_A^r = E(S_A(U_A^r)), K_A^{r-d}, S_A(V_A^{r-d}) \quad (3.4)$$

Using pipelined rounds does not significantly change our basic protocol:

1. Signatures of updates are sent f times per second, where f is the update frequency, instead $1/R$ times per second, where R is the round duration.
2. Plain-text updates and vote vectors for round occurring at time t are not sent until the round occurring at time $t + R$, where R is the round duration.

A dependency exists between the end of the round that the signature of an update is sent out and the beginning of the round that the plain-text update is sent out (see Figure 3.3). Similar to a dependency in a processor pipeline where we must wait until the dependency has passed to execute a new instruction, we must wait until the round containing the signature has passed before we can send the plain-text update. For example, if a round starts at $t=80\text{ms}$ and the round duration is 120ms , then the plain-text update must not be sent until $t=200\text{ms}$.

As the sending rate of updates increases, the real-time responsiveness and visual smoothness of the game increase. Pipelining allows us to increase the update frequency at the expense of sending more data. A game which needs a high update frequency can achieve this through pipelining without the need to artificially lower the round length. For example, a game may send 20 updates per second. This would require a round length of 50ms , or approximately a 100ms round-trip time between peers. However, if the game can tolerate a playout latency of 200ms , NEO can use a pipeline depth of 2 so that 20 updates are still sent per second while the round length is 100ms .

3.5 Performance Experiments

Our experiments are divided into three parts to test NEO's viability as a game networking protocol. In the first set of experiments, we compare NEO to Baughman and Levine's

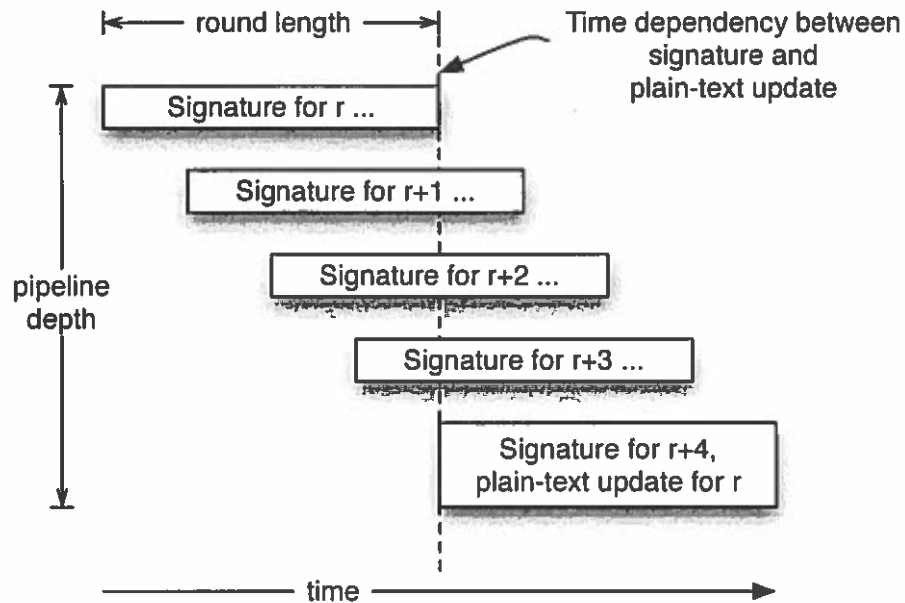


FIGURE 3.3: *Pipelining rounds in NEO:* The plain-text update cannot be revealed until the round that its signature was sent has completed.

Lockstep protocol [3]. We measure the playout latency and update frequencies of both NEO and Lockstep given packet loss and delay over the network in order to demonstrate that Lockstep is not a playable protocol because packet loss and delay cause unacceptable playout latencies and update frequencies. After this comparison, we no longer consider Lockstep in our simulations.

In the second set of experiments, we examine the effects of NEO group sizes on update frequency, playout latency, and rollback. We show that as we increase the group size, the playability of NEO only decreases slightly. This is because the amount of data sent by a NEO player increases proportionally with the number of players in the group. However, given sufficient bandwidth, NEO group size does not effect the playability of the protocol.

In our last set of experiments, we explore NEO's scalability by giving NEO an average data payload size and bandwidth and plotting the resulting NEO group size attainable under those conditions. Given that new broadband Internet connections are scaling to several megabits per second, NEO will easily scale to over 100 players.

3.5.1 Methodology and Metrics

We developed NEO and Lockstep on top of the *ns-2* simulator [72], using packet level traces and logs to measure the capabilities of both protocols. *ns-2* is a widely used packet-level network simulator that allows us to study the effects of dropped or delayed packets on the performance of the protocols.

For all of our simulations, we used an Internet topology where a central node acts as the Internet cloud connecting all other nodes. A diagram of the topology is in Figure 3.4. More complex topologies would have introduced changes in the distribution of packet loss and latencies, but would have had little effect on the protocols since neither Lockstep or NEO is sensitive to these effects.

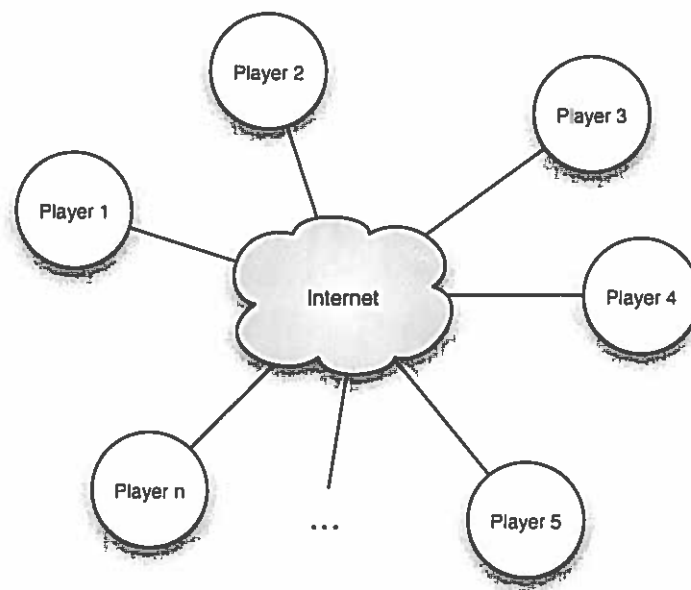


FIGURE 3.4: *NEO simulation topology:* For our simulations, we connected 1 to n players together in a NEO group, each connected to the Internet cloud. Latencies from the players to the Internet varied by experiment.

We use three metrics to measure NEO's real-time responsiveness: update frequency, playout latency, and rollback. As discussed in Section 3.4, the update frequency is a measure of how many updates are received each second. The playout latency is a measure of the delay from when an update is sent to when it can be committed for the player. Rollback

has two facets, the number of rollbacks that occur and the length of time that a player must rollback. Expected values for these metrics are listed in Table 3.3.

3.5.2 NEO and Lockstep

Our first set of experiments compare NEO and Lockstep by measuring the update frequency and playout latency of both protocols when they experience increased delay and packet loss on the network. Because Lockstep is the only other protocol resistant to protocol-level cheating, we must compare both protocols' playability. If Lockstep is playable, then NEO is not needed for distributed, multiplayer games.

Our hypothesis is that Lockstep is not a *playable* protocol, as defined in Section 3.1, because of its stop-and-wait design which will cause it to have intolerable update frequencies and playout latencies when it experiences packet loss and increased delay. NEO, on the other hand, relies on majority voting and will be playable under adverse network conditions. To test our hypothesis, we designed two experiments. Using a group of Lockstep players and a group of NEO players, we simulated packet exchanges over our simulation topology and inserted packet loss and increasing delay over a single link. Packet loss and increased delay would occur, for example, if a player began experiencing congestion over the network.

Playout Latency and Increasing Delay

Our first experiment was designed to test the affects of packet delay on the playability of NEO and Lockstep. We simulated a group of players communicating over the Internet using our topology from Figure 3.4 and we increased the latency of a single player from 20ms to 1000ms. All other players had a 20ms delay to the Internet. In addition, all players had sufficient bandwidth so that bandwidth limitations would not affect our results. This means that the minimum delay between any two players is 40ms, but may be higher due to the increased latency of our chosen player. Table 3.4 lists our experiment paramters.

Figure 3.5 plots the playout latency of NEO, Lockstep, and desirable playout latencies of FPS, RPG and RTS games archetypes (taken from Table 3.3). Note that in this figure, lower playout latencies are better. NEO is able to maintain a playout latency that is suffi-

TABLE 3.4: *Experiment parameters for NEO with increasing delay.*

Delay from Internet to players:	20ms
Increasing delay from Internet to one player:	20ms – 1000ms
NEO round length:	50ms
NEO pipeline depth:	1
Round length and pipeline depth adjustment:	none

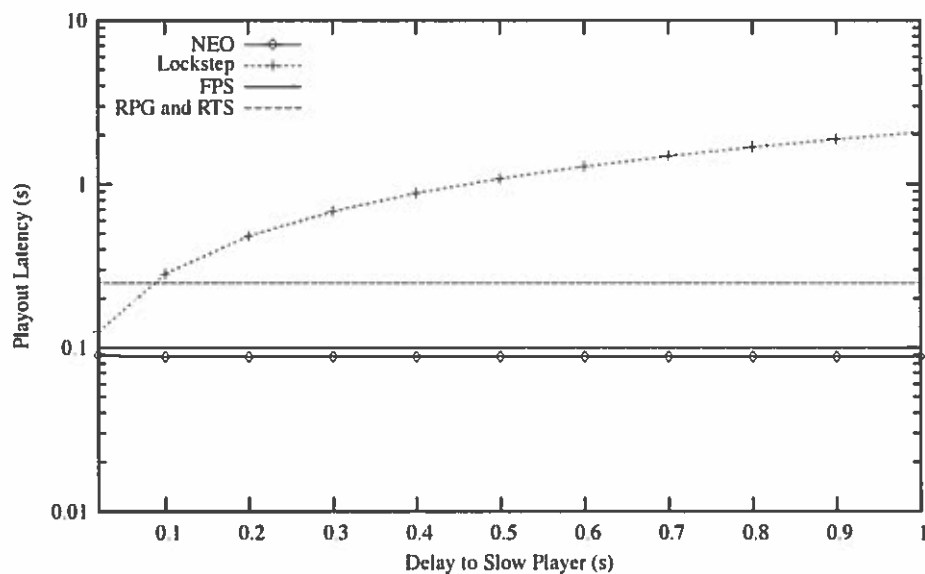


FIGURE 3.5: *Playout latency of NEO and Lockstep:* NEO maintains a playout latency suitable for FPS, RPG and RTS game archetypes (lower is better). Lockstep's playout latency depends on the maximum latency between any two players, therefore the increased delay of a single player prevents it from achieving a low enough playout latency for FPS. Furthermore, once the maximum latency to any player exceeds 80ms, it is no longer playable for RPG and RTS games.

cient for all game archetypes, while Lockstep cannot. In fact, Lockstep's playout latency will always be 3 times the maximum latency between any two players due to its use of reliable transport. In this case, our single player with increasing latency causes all players in the game to have a poor playout latency.

NEO can maintain a low playout latency because of its design. The majority voting mechanism frees the group from having to wait for the updates from players who might have late packets because unless an update was received on time by a majority of players, it will simply be ignored. With NEO, the playout latency increases only until the single

player is beyond the maximum round length. At this point, the playout latency drops back down to its expected value, about 2 times the round length. Once the single player is beyond 100ms, her moves are dropped by the other players.

Update Frequency and Increasing Delay

In our second experiment, we look at the update frequency of NEO and Lockstep with increasing delay. Using the same simulation setup (Table 3.4), we measured the number of updates per second to understand how increased delay can affect the playability of NEO and Lockstep. Examining Table 3.3 shows that we would like our game protocols to achieve 20 updates/second for FPS-type games and 5-10 updates/second for RPG and RTS games.

The results of our experiment can be seen in Figure 3.6. This figure shows that NEO is capable of achieving the update frequency necessary for FPS, RPG and RTS games, even when a minority of players are experiencing delays of up to 1 second. On the other hand, Lockstep is not playable for any of the game archetypes; if any player has more than a 100ms delay, Lockstep only sends 3 or fewer updates per second.

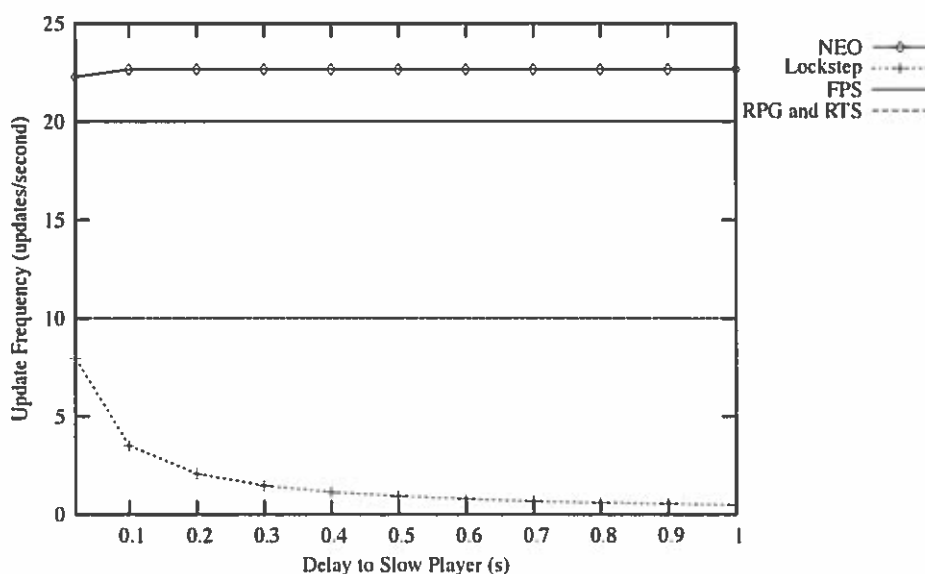


FIGURE 3.6: *Update frequency of NEO and Lockstep:* As delay is increased, NEO can maintain a high update frequency, making it capable of being used for FPS, RPG, and RTS games (higher is better). However, Lockstep is incapable of achieving the needed update frequencies for FPS, RPG, or RTS games.

TABLE 3.5: *Experiment parameters for NEO with packet loss.*

Delay from Internet to players:	20ms
Packet loss:	0 – 10%
NEO round length:	50ms
NEO pipeline depth:	1
Round length and pipeline depth adjustment:	none

NEO's superior performance over Lockstep is due to its majority voting mechanism. Lockstep must stop and wait for the verification of each update to be received by all players before sending out future updates. This means that the slowest connection between any two players determines the maximum number of updates that can be sent each second. NEO, on the other hand, uses its majority voting mechanism to decide in a distributed fashion which updates are valid. Even though a minority of the players' updates may have arrived late, NEO can still decide which updates are valid so that a majority of players send a sufficient number of updates each second.

Update Frequency and Packet Loss

In order to understand the effects of packet loss on the update frequency of NEO and Lockstep, we designed an experiment which increased the packet loss on the network from 0 to 10%. Using our Internet topology, we ran both protocols and measured the update frequencies achieved during packet loss. Table 3.5 lists the parameters for the experiment.

Figure 3.7 shows that as we increase packet loss, the update frequency of NEO remains mostly constant (though it drops slightly) while the update frequency of Lockstep drops to an insufficient level. The impact of a lost update can have a dramatic effect on Lockstep because the protocol must decide when to try to recover the missing packet. Using TCP, this could take as long as a few seconds.

NEO maintains an update frequency that supports the three main archetypes of games under a packet loss of 10% and below. Because a packet only has to be recovered if a majority of players received it, many lost packets do not need to be recovered as Figure 3.7 shows.

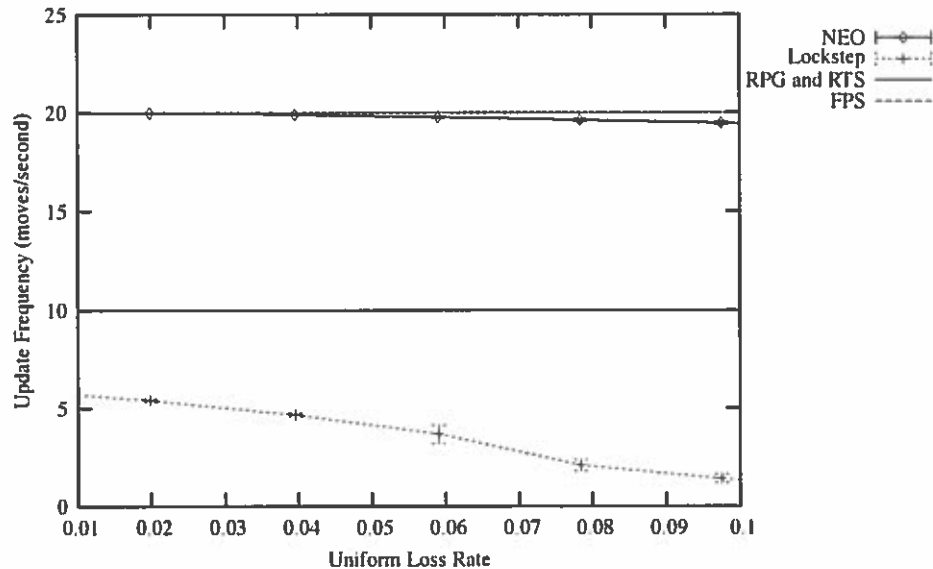


FIGURE 3.7: *Update frequency of NEO and Lockstep with packet loss:* As packet loss increases NEO maintains a high update frequency because the game can progress even when packets are lost, making it suitable for FPS, RPG and RTS game types (higher is better). Lockstep, on the other hand, has to wait for lost packets to be recovered, and therefore cannot maintain a high update frequency.

Lockstep's update frequency is insufficient for supporting the main multiplayer game archetypes when packet loss occurs. This is due to the fact that when a packet is lost, all players must stop and wait for it to be recovered before continuing the game.

Playout Latency and Packet Loss

Our fourth experiment was designed to compare the playout latency of NEO and Lockstep when they experience packet loss. Our hypothesis is that NEO will be able to compensate for packet loss and maintain a sufficiently low playout latency for all game archetypes, while Lockstep's playout latency will suffer because of its stop-and-wait mechanism.

Using the topology from Figure 3.4 and the parameters from Table 3.5, we simulated a group of NEO and Lockstep players communicating over the Internet while experiencing an end-to-end packet loss between 0 and 10%. Figure 3.8 shows the results of this experiment. The error bars on the graph display the 95% confidence interval on the average taken.

Lockstep demonstrates two trends. First, it consistently has a worse playout latency than NEO. Second, its round lengths fluctuate wildly under packet loss. This trend is due

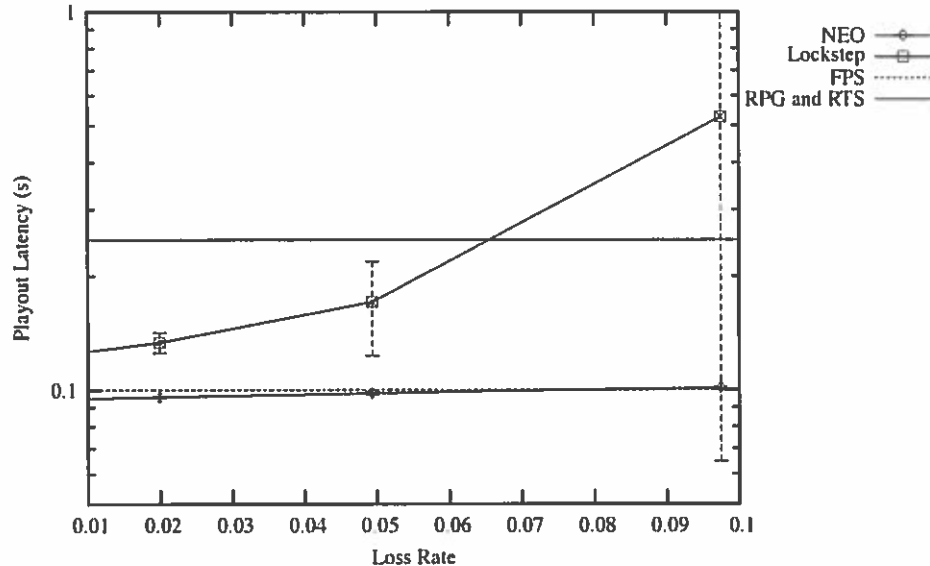


FIGURE 3.8: *Playout latency for NEO and Lockstep with packet loss:* NEO consistently remains playable for FPS, RPG, and RTS type games with very little variation in the playout latency. Lockstep has a large variation in playout latency and is only capable of being playable for RPG and RTS style games when packet loss is below 7%.

to the use of reliable transport which must determine when a packet has actually been lost. End-to-end packet loss that exceeds 7% will cause Lockstep to be unplayable for FPS, RPG and RTS game archetypes.

NEO performs well, keeping its playout latency fairly constant for all players, even in the face of packet loss. NEO performs so well even under higher packet losses because its voting mechanism can mask those losses. If a packet is received by a majority of players, then only the minority of players which didn't receive it are required to recover it. Therefore, only a minority experience a larger playout latency for those updates.

Discussion of NEO vs. Lockstep

Our experiments show that NEO, even with packet loss and increased delay, remains playable for the three main game archetypes. NEO is capable of sending a sufficient number of updates within the time limits required by FPS, RPG and RTS games.

Our experiments also show that Lockstep is not playable for FPS, RPG, and RTS games given the same network conditions that NEO was subjected to. If the one-way latency

between any two players exceeds 40ms or if the packet loss on any link exceeds 2 or 3%, a game using Lockstep will quickly become unplayable.

One might argue that we can modify Lockstep to drop the player with the large latency. However, this is not a trivial problem because a player may be experiencing transient congestion leading to delay and loss. Dropping players and having them rejoin could also lead to instability in the protocol and exacerbate network conditions with an influx of traffic generated by joining the group. NEO handles this problem by ignoring delayed or lost packets while preventing cheating so that other players can continue to play the game.

NEO bounds the playout latency by the round length. If a player experiences temporary congestion, and therefore increased delay above the round length, then updates generated by the player will likely be dropped by the other players. Moreover, the player only needs to have her delay less than the round length for a majority of players (i.e., more than 50%). Finally, the other players are not affected by the latencies of those with large delays. Indeed, unless a majority of players are experiencing delay above the round length, then the protocol will progress smoothly for a majority of players.

In the case of packet loss, Lockstep suffers because it must predict when a packet is lost and when it is just late. TCP and other reliable protocols have a similar problem and have been the subject of research for years. If we shorten the time that Lockstep uses to determine that a packet is lost, packets which are just late may be incorrectly thought to be lost, with the result that Lockstep will generate additional traffic for packet recovery.

NEO uses the round length to determine if an update is lost. The difference between NEO and Lockstep is that an update in NEO does not necessarily need to be recovered, unless it was received by a majority of players. Thus, many packets that are lost are not recovered. Furthermore, rounds continue even as the update is recovered.

3.5.3 NEO Group Performance

Now that we see that NEO is capable of remaining playable given packet loss or increased delay, we examine the effects of group size on NEO performance. We simulate between 10 and 100 players over the simulation topology from Figure 3.4, and add between 0 and 10% packet loss to the network. We examined three metrics of playability:

update frequency, playout latency, and rollback. The parameters for our experiments are listed in Table 3.6.

TABLE 3.6: *Experiment Parameters for varying NEO Group Sizes.*

NEO group size:	10, 25, 50, 75, 100
Delay from Internet to players:	20ms
Packet loss:	0 – 10%
NEO round length:	50ms
NEO pipeline depth:	1
Round length and pipeline depth adjustment:	none

Neo Group Size and Update Frequency

In our first experiment, we measure the update frequency of NEO given a group size of 10, 25, 50, 75, or 100 players. Our hypothesis is that NEO group size would not significantly affect the performance of the protocol assuming each NEO player had sufficient bandwidth.

Figure 3.9 shows that NEO group size does not significantly affect the update frequency of the protocol. The downward trend of each line is due to the increased packet loss, which causes some updates to be recovered and reduces the update frequency (since those updates are lost). In all cases, the average update frequency is higher than 20 updates/second, desired by FPS games and higher than 10 updates/second desired by RPG and RTS games.

Neo Group Size and Playout Latency

We next studied the effect of group size on the playout latency of NEO. Our hypothesis is that NEO group size would not significantly affect the performance of the protocol, assuming each player had sufficiently low latency to meet the round length. Parameters for the experiment are listed in Table 3.6.

Figure 3.10 shows that NEO is capable of maintaining a sufficiently low playout latency even with 100 players and 10% packet loss. As with the update frequency experiments, the upward trend of the data lines is due to the increased packet loss, and not the increased number of players.

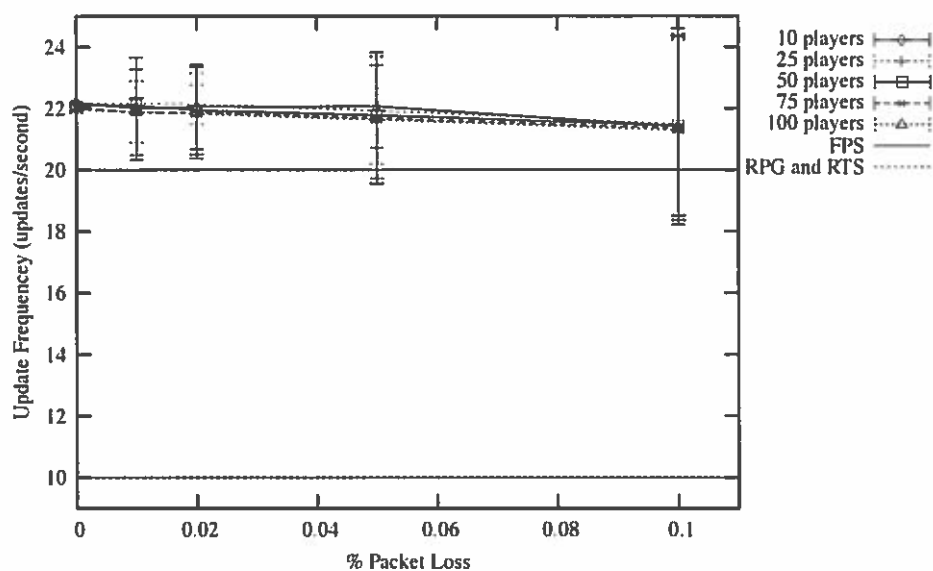


FIGURE 3.9: *Update frequency of various NEO group sizes with packet loss:* NEO's update frequency is not significantly affected by increasing NEO group sizes even with a packet loss of up to 10% (higher is better), though the downward trend of each line is a result of increasing packet loss. Note that with FPS games, we hope to achieve 20 updates/second, while with RPG and RTS games, we hope to achieve 10 updates/second.

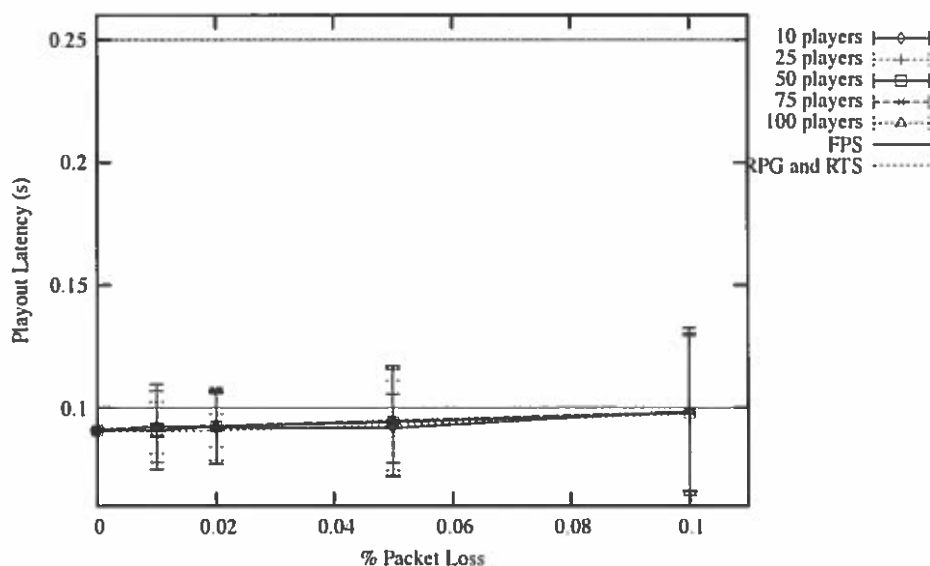


FIGURE 3.10: *Payout latency for various NEO group sizes with packet loss:* NEO's average payout latency is not significantly affected by increasing NEO group sizes with packet loss (lower is better). Note that with FPS games, we hope to achieve a payout latency of 100ms or less, while with RPG and RTS games, we hope to achieve a payout latency of 250ms or less.

NEO Group Size and Rollback

In our final set of experiments with NEO group sizes, we examine the effects of NEO group sizes on rollback. Rollback occurs when a late update arrives, but is valid. In this case, the simulation must rollback time to when the late update occurred, insert the update, and instantaneously replay all successive events to the current time in order to maintain consistency with the other players. Two aspects of rollback must be measured: how much rollback occurs as a percentage of total updates, and how large a rollback is on average. The parameters for our experiments are listed in Table 3.6.

The results from our simulations can be seen in Figure 3.11 and show that rollback increases as packet loss increases, though group size seems to have little effect on the amount of rollback that occurs. In fact, our results show a trend that the amount of rollback is approximately $1/2 * packet_loss$. This trend may be a result of the use of uniform packet loss. Since each player unicasts their update each round in NEO, at least half of those unicast packets would need to be lost in order for the update to be ignored. Thus, rollback increases linearly with packet loss.

One difficulty in designing this experiment, is that no research has been done to date that measures how much rollback is acceptable by players of FPS, RPG, and RTS games. The lines in Figure 3.11 representing FPS, RPG, and RTS games are guesses on what acceptable rollback might be. For a FPS game, where reflexes are important to the outcome of the game, we assume a 3% rollback (1 in 33) would be the maximum tolerable rollback. For RPG and RTS games, we assume that a 5% rollback (1 in 20) would be the maximum amount of rollback. Figure 3.11 shows acceptable amounts of rollback for all genres when the packet loss is less than 6%.

In addition to the percentage of updates that must be rolled back, we also measure how far we have to rollback the updates. Smaller rollbacks can be easily masked by the game through dead-reckoning, or ignored altogether if a more recent update overrides a late update.

Figure 3.12 shows that the number of players appears to have no effect on the amount of rollback that occurs. The deviation from the average increases as packet loss increases, but in general, rollback is less than one round length.

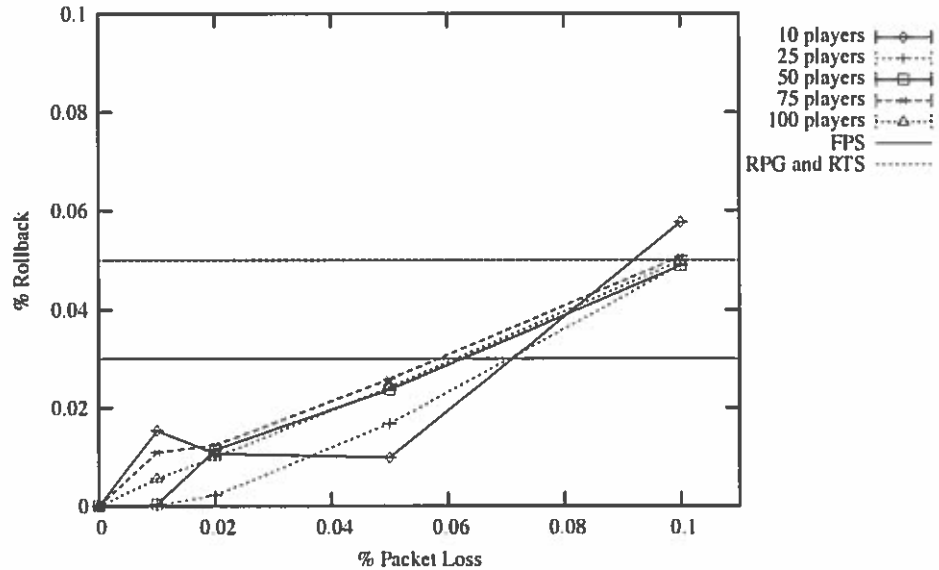


FIGURE 3.11: Rollback required as NEO group size increases: Group size appears to have little effect on the percentage of rollbacks that occur, but increases with packet loss (which is expected). The lines representing FPS, RPG and RTS games are guesses on the percentage of rollbacks tolerated.

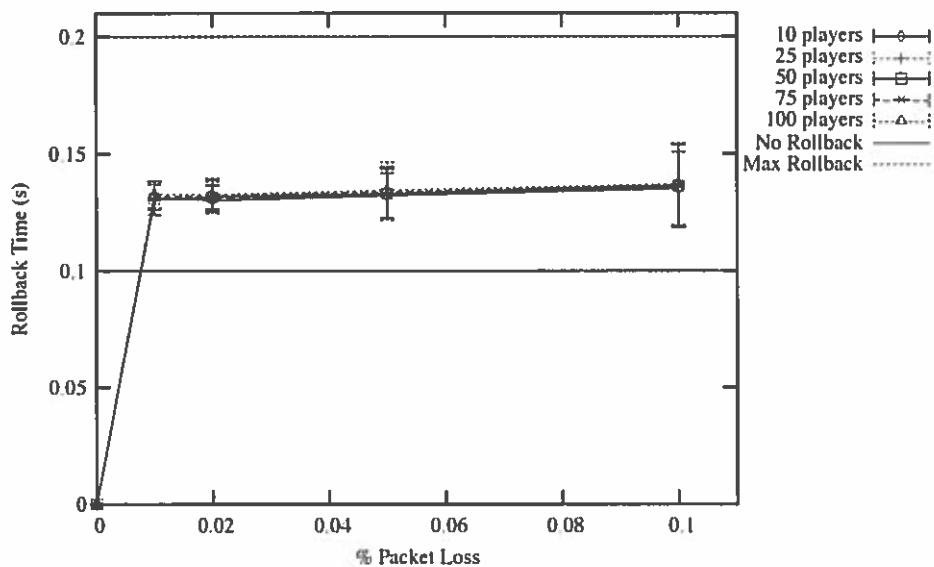


FIGURE 3.12: Length of rollback as NEO group size increases: Group size appears to have little effect on the amount of rollback that occurs. The 'No Rollback' line indicates the maximum amount of time an update can arrive without being considered 'late'. 'Max Rollback' is the maximum amount of time that any update will be late since it can the update can be routed through another player.

3.5.4 How Well Does NEO Scale?

In our final analysis of the NEO protocol, we investigate the scalability of NEO. We look at how large NEO groups can be given bandwidth restrictions, average packet sizes, and update frequencies. Given n players, we know that NEO packets are $O(n^2)$ in size. In our analysis, we answer what this means in terms of today's bandwidth and conclude that NEO can support up to 64 players over a 1Mbps link with 10 updates/second and up to 128 players over a 1Mbps link with 5 updates/second.

Methodology

We performed an analysis on NEO's scalability by first developing a formula that relates bandwidth to group size and packet payload sizes. Given n players and a payload size of m , the size of a single NEO packet is $p = m + n$. Each round, a player sends $n - 1$ packets of size p (to every other player) and receives $n - 1$ packets of size p (from every other player), thus the total required bandwidth is:

$$\begin{aligned}
 B &= (n - 1) * p + (n - 1) * p \\
 &= 2np - 2p \\
 &= 2n(m + n) - 2(m + n) \\
 &= 2(mn + n^2 - m - n)
 \end{aligned}$$

B represents the amount of data sent for each update. However, the amount of data that NEO sends also depends on the number of updates sent each second. Thus, if f is the frequency of updates each second, then a single NEO peer will need to $f\dot{B}$ bytes each second.

By solving for n , we can determine how much bandwidth is required given a NEO group size of n . Thus, we get⁶:

$$n = \frac{1}{2}(1 - m \pm \sqrt{1 + 2B + 2m + m^2})$$

⁶We ignore the negative root of this equation since we cannot have fewer than 0 players.

To determine the payload size, we look at previous measurement studies of multiplayer games that examine the packet sizes produced by players. In one study, the mean packet size was 40 bytes [73] while in another study, measurements showed that over 97% of all packets were 31 bytes or less and 99% of the packets were 50 bytes or less [74]. Thus, we chose a range of 20 to 100 bytes.

We distinguish between packet size and payload size because with NEO, the packet size changes, depending on the number of players and how many updates we wish to send each second. The payload size, as shown in our graphs, is only the amount of data relating to the game that is transferred with each packet and is independent of the number of players in a NEO group.

Update frequency was determined by the archetype of the game. If we look at Table 3.3, we see that each type of game has a maximum latency tolerated. The inverse of this value gives us the frequency of updates we would need to send to achieve that latency. Of these, FPS and sports games have the most strict requirements and given the limit on human reaction time, we use 10 updates/second as the maximum frequency that any game would send need to send updates. A more common update frequency would be 5 updates/second, covering most other classes of games. As such, we provide two graphs: one measured at 10 updates/second and the other measured at 5 updates/second. We expect the performance of NEO to lie somewhere within that continuum.

Scalability Results

Figures 3.13 and 3.14 plot the results of our scalability analysis of NEO when we use a payload size between 10 and 100 bytes and when we increase the bandwidth from 1Kbps to 10Mbps. Figure 3.13 demonstrates NEO group sizes when NEO is configured to send 10 updates/second—a rate sufficient for FPS games. Group sizes are represented by lines and are ordered from left to right on the graph.

Figure 3.13 shows two things:

- NEO can support up to 64 players over a 1Mbps link and 128 players over a 7Mbps link (a typical cable broadband download speed).
- The bandwidth required by a NEO peer increases as the NEO group size increases.

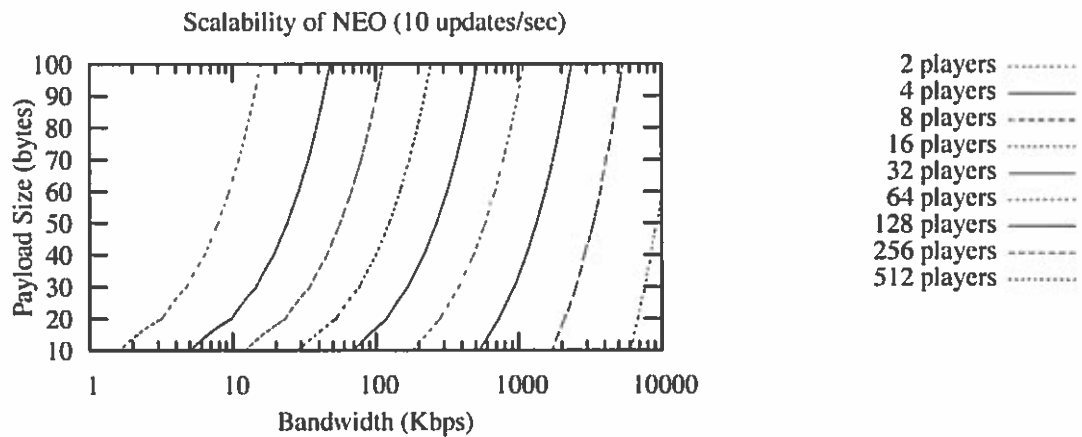


FIGURE 3.13: *Scalability of NEO with 10 updates/second:* In this graph we see that the payload size significantly effects the scalability of NEO. Given a 1Mbps link, NEO can support up to 64 players which would be typical of FPS games. At 7Mbps, NEO supports up to 128 players.

Figure 3.14 is a plot when NEO is configured to send 5 updates/second, which is a rate usable by RPG and RTS games. Group sizes, beginning with 2 players, are shown from left to right. This figure shows that NEO can almost support 128 players with a 1Mbps and can support up to 256 players with a 7Mbps link.

Scalability Discussion

NEO's scalability is limited by its packet size, which is $O(n)$, where n is the number of players in the game. Given that a player must send out $n - 1$ updates to the other players, a single player generates network traffic that is $O(n^2)$. However, the typical payload of a game packet is between 10 and 100 bytes. Given today's broadband connection speeds, NEO can support games between 64 and 128 players.

Note that in our analysis, we calculate the total bandwidth required by a peer. A typical Internet connection has separate upload and download speeds. For example, in Figure 3.13, we see that with a 7Mbps link, we can support up to 128 players with 10 updates/second. This requirement is split evenly between upload and download speeds (NEO sends $n - 1$ updates and receives $n - 1$ updates each round). Thus, our Internet link would need to have

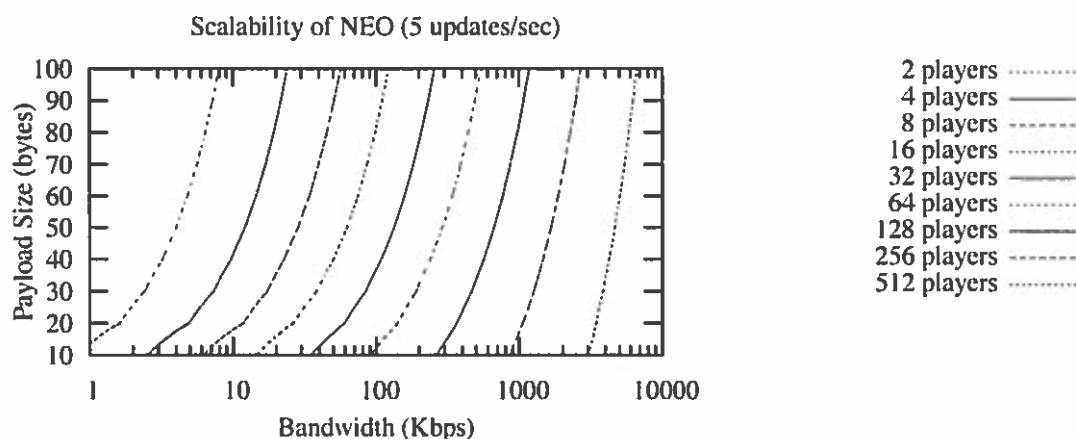


FIGURE 3.14: *Scalability of NEO with 5 updates/second:* In this graph we see that by reducing the number of updates/second to 5, we can effectively double the number of players supported at a given bandwidth; at 1Mbps NEO supports 128 players while at 7 Mbps it can support up to 256 players. This reduced update frequency is more typical of RPG and RTS games.

a 3.5Mbps upload/3.5Mbps download speed. In the next chapter, we discuss how to scale cheat-proof event ordering beyond the small groups that NEO is capable of.

3.6 Conclusion

Scalable, cheat-proof and real-time event ordering is a difficult problem that needs to be solved in order to realize peer-to-peer MMOGs. NEO solves one piece of this problem: cheat-proof and real-time event ordering. To achieve this, NEO uses a majority voting system with cryptography and maintains majority game consistency between peers. This prevents peers from cheating while maintaining consistency as our proofs demonstrate.

Our analysis and simulations show that while NEO does indeed have significant overhead, NEO is capable of achieving a sufficient update frequency and low playout latency while preventing rollback that would occur if all updates had to be accepted. NEO trades bandwidth for real-time responsiveness. However, we feel that this tradeoff is acceptable due to the fact that bandwidth on the Internet continues to increase exponentially while

latency between hosts has a theoretical minimum of the speed of light and cannot be significantly reduced in the future.

We investigated traditional concepts of consistency in distributed systems. Strict consistency and Lamport's sequential consistency are two models easily adapted to the needs of multiplayer games. As a result, we created strict game consistency, where a single, global sequence for all events is seen by all players. This model is particularly well suited for client/server architectures, where the server orders all incoming events and informs the clients of the result of that ordering. We also created sequential game consistency, which is analogous to Lamport's sequential consistency. In this model, the sequence of events is an ordered, interleaving of events generated by the players. Baughman and Levine's Lockstep protocol [3] achieves sequential game consistency.

We created a new consistency model for games called *majority game consistency*. In this model, the sequence of events only includes those events seen by a majority of players. The advantage of this model is that it allows us to mask failures and delayed updates by not waiting to include them in the final sequence of events. However, we can also extend this consistency model and apply it towards traditional distributed systems:

Definition 3.6.1. *Majority consistency:* the result of any computation is the same as if all reads and writes follow a sequential order, and only the reads and writes seen by a majority of systems appear in that sequence in the order specified by its program.

We expect that this model of consistency for distributed systems would have similar performance benefits to NEO when used over best-effort networks such as the Internet.

In summary, our work with NEO has two important contributions:

1. We have expanded the traditional notions of consistency and applied them towards games in the form of majority game consistency.
2. We have developed the NEO protocol, which addresses both low-latency event ordering and protocol level cheats. We analyzed NEO with an extensive set of simulations to show that its performance is superior to prior work and will scale depending on the available bandwidth of the players. We also proved its resilience to cheating.

As discussed previously, NEO has limitations with its ability to scale. Under present-day bandwidth levels, such as DSL, NEO can support up to 128 players over a 1Mbps link. In the next chapter, we describe the N-Trees architecture and the notion of scoped event forecasting which, when combined with NEO, provides a scalable communication architecture for cheat-proof and real-time event ordering.

CHAPTER IV

N-TREES

4.1 Introduction

While NEO provides cheat-proof, real-time event ordering for peer-to-peer game architectures, its scalability is limited by the incoming and outgoing bandwidth of the majority of peers in the game. This limitation is due to the fact that NEO requires all-to-all communication. If we relax this constraint and still provide the security and real-time message delivery of NEO, our system will be capable of scaling to a larger number of peers. Accordingly, we have created a hierarchical structure to organize peers, called an N-Tree, that relies on NEO to provide cheat-proof event ordering within NEO subgroups in the game while allowing events to be propagated quickly to affected subsets of peers in the game.

Previous work in scalable peer-to-peer games has used some form of application layer multicast (ALM), such as the publish/subscribe system of Mercury [4] or region-based multicast [6], which used Scribe over Pastry [14, 54]. These systems organize their dissemination paths based on the proximity of peers on the network, instead of the proximity of peers in the virtual world. The result is that peers which are close in the virtual world, but far apart in the network, will have to route messages over potentially long distances in the underlying ALM structure to order events. Even though some of these systems guarantee $O(\lg n)$ application layer hops with n peers, even a few additional hops can add considerable delay, making interactive event ordering infeasible.

As discussed earlier, traditional event ordering protocols for distributed systems, such as the Paxos algorithm which only tolerates stopping failures, can require up to five rounds

of communication [17] and do not scale. In fact, even without faults, distributed event ordering requires at least $\Omega(n^2)$ messages per event and two rounds of communication between all peers [29].

N-Trees face the same challenges as did NEO. First, how do we order events and ensure consistency between players? Second, how do we deliver messages in a timely manner and keep the game responsive in real time? Third, how do we prevent protocol-level cheating?

N-Trees address these problems through a hierarchical structure and *scoped events*, which help determine which subset of peers should receive an event. N-Trees are a generalization of the *octree*, from computer graphics [75], that recursively subdivide an N-dimensional space. For event ordering, the N-Tree subdivides the *application state space*, which is the application domain that all peers share and modify. In a peer-to-peer game, the application state space *is* the virtual world; hence, the state space of computer games can be naturally decomposed into an N-Tree.

Peers are organized into an N-Tree by their scope of interest in the virtual world. The N-Tree allows peers to efficiently move to new locations, discover other peers that are in close proximity, and propagate events to other parts of the virtual world. By joining the N-Tree, peers know which peers are close by, and can therefore order events directly with those peers without having to exchange events with other, further peers.

Peers generate scoped events, which are events that are labelled with a tuple representing the scope of impact that the event has in the virtual world. When a peer generates an event, it uses the N-Tree to propagate the event to other peers within the event's scope. If the scope of the event is contained within the leaf node, then the peer only has to exchange events directly with other peers in the leaf. By routing scoped events over the N-Tree, we minimize the number of peers that are involved in event ordering. Finally, we develop a new technique called *forecasting* to minimize the amount of rollback and determine which events are valid in the N-Tree.

In this chapter we show that this architecture, which builds on NEO and uses N-Trees, ensures that events are delivered in a timely manner to peers who need them while allowing peers to move efficiently through the tree as they move in the virtual world. We formally define the notion of majority game consistency as it relates to N-Trees and scoped events and show that N-Trees satisfies this new notion of consistency. We show the results of an

initial measurement study that looks at the characteristics of virtual populations in order to create a model for MMOGs. With further measurements, this model can be used for future simulations and reasearch. In addition, we have verify the performance of N-Trees through a set of simulations that examine the height of the trees based on our simulation model. Our results show that the maximum height is reasonable even with a non-uniform distribution of players in the virtual world.

4.2 Related Work

N-Trees are designed to add scalability to the cheat-proof and real-time event ordering that NEO provides. Prior research in this area has addressed scalability or consistency, but has not addressed the problem in conjunction with cheating and low-latency event ordering.

We are interested in N-Trees because they are optimized for event propagation so that the fewest number of nodes are contacted for event ordering. Existing peer-to-peer structures, such as Gnutella, Chord, CAN and Pastry [56, 12, 11, 54], are optimized for fast storage and retrieval of data using two functions, *insert(key, value)* and *lookup(key)*. DHTs do provide the kind of mapping we need for the application space – we might map the application space to the key space of a DHT, for example. However, propagating events to neighbors and relocating to new places in the DHT would be too slow for scalable event ordering.

N-Trees have a similar advantage when compared to using application layer multicast (ALM) such as Bayeux, CAN-multicast, Narada, NICE, and Scribe [23, 22, 21, 24, 14]. Multicast is optimized to send as few messages as possible in one-to-many (or many-to-many) communication. However, multicast trees are built based on end-to-end delays between hosts, not on the interests of group members. While multicast reduces messaging, all messages are sent to all group members and filtering messages to relevant members is not a trivial task. Further, in a peer-to-peer game, every member sends messages, requiring a shared tree (such as HMTP [76]), or n source-specific trees. On the other hand, ALM could be used in conjunction with N-Trees between leaders of each node in the tree, and even between nodes in each small group, to reduce messaging.

Prior research in scalable, peer-to-peer games has tried to address scalability, but has not considered the problem of cheating. Mercury, for example, provides a publish/subscribe architecture to support massively multiplayer online games (MMOs) [4]. In this architecture, a multi-attribute query language is used so that game state is published in Mercury and players receive a small subset of state changes according to their subscriptions; thus, the total state that each player receives is reduced by Mercury. The primary difference is that Mercury uses a DHT to store and route information while N-Trees build a domain-specific tree. Joining an N-Tree at a particular node determines what data a peer receives, whereas with Mercury the subscription determines what data is routed to a peer.

Knutsson et al. statically divide a world into *regions* and use Scribe to multicast messages to all members of a region [6]. Peers join the multicast tree of the region they are interested in, but this also means that peers will have to forward unrelated traffic. This occurs because peers are members of the underlying Pastry DHT and will likely be on the path of other multicast trees (since a source-specific tree is built for each region). Further, the static division of the virtual world limits the scalability of the system if some areas suddenly become popular. N-Trees in this case could be used in each region to increase the scalability of their work.

Baughman and Levine proposed the first peer-to-peer protocol, called Asynchronous Synchronization (AS), that was built upon Lockstep to prevent the *timestamp cheat* [3]. However, AS is subject to both the *fixed-delay* and *suppressed update* cheats. Further, simple collusion allows players to execute the *timestamp cheat*¹. Furthermore, AS suffers from a long payout latency when a single player begins to have significant packet delay or loss.

Recently St. John and Levine designed Ghost as an extension to AS to handle variable delays and partitions in the network [77]. Ghost handles consistency issues due to packet loss or delays by allowing game states to diverge and grouping those players with the same game state. Both Ghost and AS scale by being used with a statically or dynamically divided virtual world, such as with Knutsson's work [6], or at the leaves of N-Trees.

¹The timestamp cheat occurs in the following manner: Player A receives a plain-text update from Player B and forwards it to Player C before Player C should have received it. This allows Player C to alter her next event to take advantage of the knowledge of Player B's update.

The idea of subdividing the virtual world to increase scalability has been around since the 1990's, when researchers were investigating Distributed Interactive Simulations [78, 79]. However, this related work focused on dividing the virtual world with a grid to form cells, with each cell being assigned a multicast group. Thus, joining a cell meant joining a multicast group to communicate with other members of the cell. The work by Knutsson et al. uses a similar approach over a DHT [6]. N-Trees differ significantly from this prior work because they dynamically divide the virtual world into a hierarchy, they are used to propagate events, and they are used to order events between players.

4.3 Definitions

Definition 4.3.1. *N-Tree*: An N-Tree is defined inductively as follows:

(Base Case): a Leaf without children

(Inductive Case): a Node with 2^N children, each a distinct N-Tree

An N-Tree is a generalization of *octrees* with N dimensions instead of only three that an octree has [75].

In our use of N-Trees, the leaves of the N-Tree are NEO groups. Each NEO group elects a leader to act as the node in the N-Tree and nodes communicate with each other with reliable communication. All events are generated at the leaves of the tree and propagated through the tree, as discussed in Section 4.4.

Definition 4.3.2. *Application state space*: The domain of a distributed application that is hierarchically structured such that a domain has 2^N sub-domains. Each sub-domain shares this domain as a sole parent and can be recursively subdivided ad infinitum. We place the restriction that each division of a sub-domain has 2^N children solely for the purpose of mapping the application state space into an N-Tree. The term *state space* is used interchangeably with *application state space*.

In a peer-to-peer game, the application state space is the virtual world and its contents, including the players of the game. Events occurring in the virtual world include player movement, taking an item in the virtual world, and larger events such as a snow storm in an

area. The state space in this example includes two or three *dimensions*, or variables, which correspond to the actual dimensions in the virtual world. Each object in the virtual world has a corresponding location, and perhaps volume, in the virtual world.

Definition 4.3.3. Scoped event: A tuple consisting of an action, the location of the action, and a function representing the scope of impact of that action in the application state space. The location of the event indicates where the event originally occurs while the scope indicates how broadly the event can impact the application state space.

For example, taking a treasure in a game creates an event that occupies a single point in space, which is the location of the treasure. A snow storm, on the other hand, would have a scope equal to the maximum radius of the storm. In this case, the scope is a circle centered at the location of the action, with a given radius. Scopes can be defined by an arbitrary spatial function which defines the impacted area in the virtual world.

In Figure 4.1, we illustrate a 2-tree, or quadtree. In a quadtree, the state space is a Cartesian square that is subdivided as necessary to meet the scoping requirements of the application and the current peers in the network (Figure 4.1a). The resulting N-Tree is seen in Figure 4.1b. For example, the 2 dimensional virtual world corresponds to a 2-tree so that each point in the world maps to a leaf in the 2-tree. In this figure, a snow storm occurs at (.48, .28) with a scope defined as a circle with a radius of .05. If the scope of the event exceeds the boundary of the node it occurs in, it will be propagated to all nodes that intersect its scope.

4.4 The N-Tree Protocol

The N-Tree architecture organizes peers, provides a mechanism for propagating updates, and uses rollback and forecasting to maintain consistency. The N-Tree structure is maintained by the N-Tree protocol, which dictates where nodes join the tree, when leaves are divided because their population is too high, how updates are propagated in the tree, and how updates are voted on to determine their validity.

Peers in the system act as the nodes in the N-Tree and are elected to that role by a leader selection protocol. To communicate between nodes, leader nodes use reliable com-

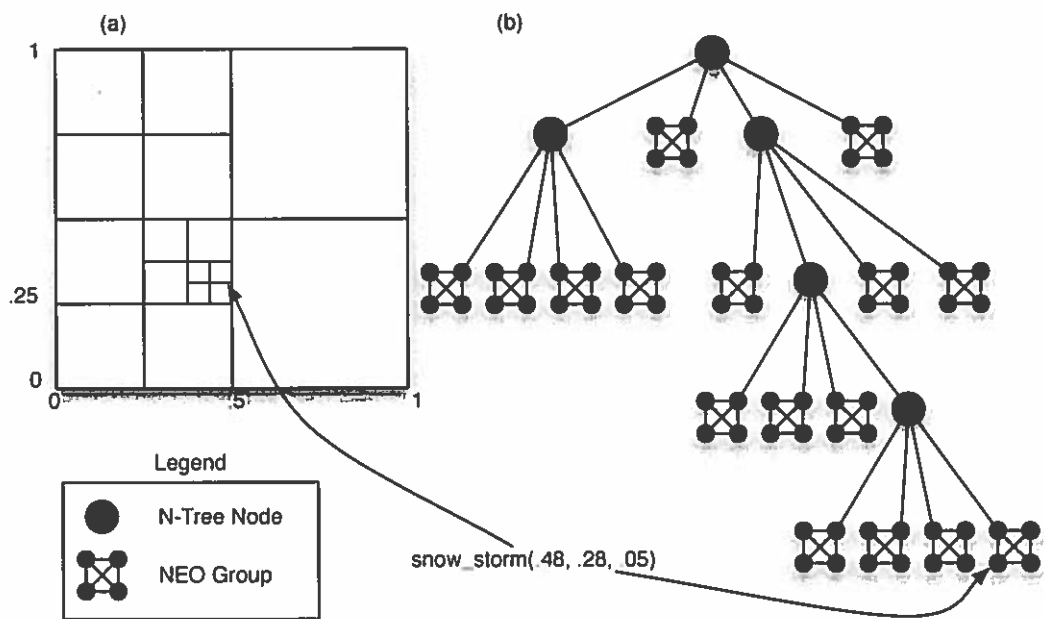


FIGURE 4.1: *Cartesian application space and quadtree representation:* In (a), the virtual world is represented by a Cartesian square, with lines representing the areas covered by leaves in the N-Tree. In (b), we see the equivalent N-Tree representation. A snow storm occurs at (.48, .28) with a radius of .05. The event occurs in the shown node and may be propagated to other nodes if the radius of the event exceeds the boundary of the node it occurs in. At the leaves of the N-Tree, groups of players run the NEO protocol for event ordering and consistency.

munication. However, allowing a peer to become a node leader enables that peer to cheat because they can purposely drop or delay packets and alter vote tallies. We address security in Section 4.6.

4.4.1 Node Leader Selection

In the N-Tree protocol, peers act as the nodes of the N-Tree for the purpose of event propagation and vote tallying. Because we are assuming that N-Trees will support a large number of peers, we know that our tree will also have a large number of nodes, necessitating a fast leader election protocol.

To elect a leader for a node in the N-Tree, we use the following method. We assume that all players have a unique ID assigned to them by a trusted authority. This authority is most likely the game developer who uses it to ensure that players are authorized to play the game. We also assume that all players use the same pseudo-random number generator which is seeded and used for leader election.

Each player will generate a pseudo-random number, hash it, digitally sign it and exchange it with other members of the NEO group using reliable communication. Once all hashes have been received, all players in the NEO will then exchange the plain-text version of their pseudo-random number. Players then use modulo addition and all the random numbers and use the result as a seed for the pseudo-random number generator. Once the generator is seeded, we pick the first result from it and the player with the closest ID to it is the leader.

To change leaders, we simply pick the next number from our pseudo-random number generator and pick the next player with the closest ID. We repeat the seeding whenever a new player joins the NEO group. This node leader selection protocol is resilient to most security attacks. Further security measures are discussed in Section 4.6

4.4.2 Event Propagation

Event propagation occurs when the scope of an update issued by a peer in a NEO group (at the leaves of the N-Tree as in Figure 4.1b) exceeds the sub-domain of a node in the N-Tree. When a peer generates an update, the update is exchanged with other peers in

the same NEO group. When the group leader receives the update, she checks the scope and compares it to the scope under her responsibility. If the scope of the update is not completely contained by her node, she forwards the update to her parent. The parent then forwards the update to all children whose scope intersects with the update scope, and additionally forwards it up the tree again.

In Figure 4.2, we illustrate how event propagation works. At the top of the figure, our virtual world is subdivided into 16 nodes of the N-Tree, with the Tree illustrated at the bottom of the figure. An update occurs at (x,y) , represented by a black dot in the virtual world. The circle around the dot represents the scope of the update and all affected nodes are shaded gray.

Events always begin at leaves in NEO groups. In the figure, the leader for the area where the update at (x,y) occurs at discovers that the scope of the update exceeds her scope

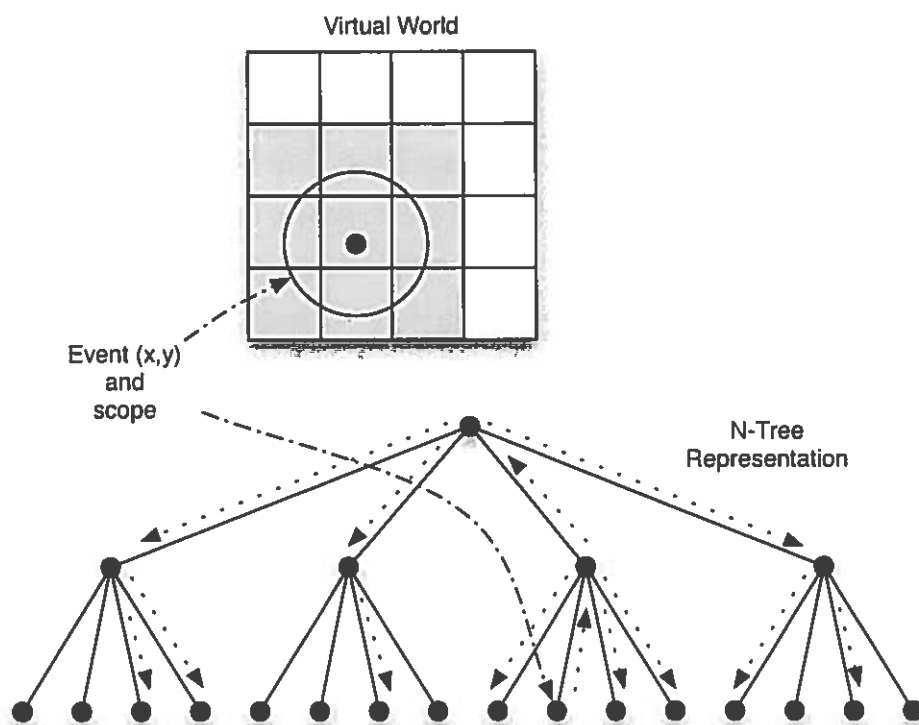


FIGURE 4.2: *Event propagation in N-Trees:* The update at (x,y) has a scope that exceeds the boundary of the node it was generated in and must propagate the update to all shaded nodes in the virtual world. At the bottom of the figure, the dotted arrows indicate the directions the update at (x,y) takes in the N-Tree to reach its intended destinations.

of control and forwards the update up the N-Tree. From here, the update is forwarded to each node within the scope of the update. Interior nodes continue to forward updates up the tree until the update reaches a node whose scope contains that of the event—at which point the update is propagated down the tree.

4.4.3 Forecasting

Propagating scoped events via the N-Tree introduces the problem that updates may experience a potentially high latency before they are committed. As the N-Tree increases in depth, the delay from when an update was generated to when it is committed increases proportionally. If the update is time-stamped with its round number according to when it is generated, all players who fall within the update's scope will receive a late update and thus will have to rollback whenever the update is finally agreed upon. In essence, event propagation through the N-Tree could cause a large amount of rollback.

We have created a new technique to minimize the rollback caused by event propagation which we call *forecasting*:

Definition 4.4.1. Forecasting: Time-stamping an update with a round number to occur at some time in the future, based on the scope of the update.

The idea behind forecasting is that updates that affect a large number of players beyond the scope of a leaf are significant updates which often (and should) take some finite amount of time to execute. Indeed, game players already easily accept such delay in games. For example, a player must occupy and defend a tower for several minutes before they control the tower as a resource. As another example, launching a missile that might damage a large continent takes several minutes before it is successfully launched and is preceded in game by a countdown and the emission of smoke and fire from the missile as it prepares to launch². Thus, forecasting is masked in the virtual world by the game mechanics for updates that have a large scope. We forecast an update with N-Trees and wait for approval of the update, during which time the game animates the beginning of the event. If we

²Consider the amount of time it takes to launch the Space Shuttle.

successfully forecast the event and it is accepted, then we will receive agreement before the completion of the update and no rollback will be necessary.

Forecasting eliminates the majority of rollback in the game because peers decide to accept or reject an update before it is scheduled to occur. If the event is not forecast sufficiently in the future, the update is rejected. However, forecasting does not eliminate all rollback. For example, a plain-text update could arrive late after it was agreed upon. The player receiving this update would then have to rollback and insert the update in her sequence of events.

One might argue that this limits the usability of large-scoped events to only those which take some amount of time to execute, and we agree with this statement. Certainly, the use of peer-to-peer networking for games, and the requirement that we prevent cheating and maintain consistency within the game has a very real cost in terms of latency. Thus, peer-to-peer networking cannot be used for every single type of multiplayer game, though we believe it will work for a large percentage of those games.

Ideally, we would forecast an event to occur at a time in the future that ensures that everyone has agreed that the event is valid. Since we cannot know when an event will be agreed upon *a priori*, we settle for an estimation of this time based on the scope of an update. The amount of time we forecast is based on the height of the tree and the scope of the event. An event with a small scope may only need to traverse up the tree by one or two nodes, and hence we would forecast based on the average latency between nodes of the N-Tree multiplied by the number of hops in the N-Tree our update must take.

To estimate how far in the future we must forecast an event so that it arrives on time, we can estimate how many nodes in the tree our event intersects with, and therefore how high in the N-Tree an event must travel to reach the leaves it intersects with. In fact, we can bound the maximum height of a 2-Tree by a constant value such as 20 because at this height, if players are uniformly distributed, the N-Tree would support several trillion players—a sufficiently large number of players for a single game. Therefore, as an upper bound, an event that affects all leaves in the tree is at most the maximum height of the tree.

Deciding how far in the future to forecast an update depends not only on the height of the N-Tree and how many leaves are affected, but also on the average latency between players. Unfortunately, few studies have measured latencies between end users at their homes,

but instead of looked at measuring properties of the Internet between well-provisioned hosts. Part of the difficulty here is that most home users are firewalled and use Network Address Translation (NAT).

4.4.4 Event Ordering and Majority Voting in the N-Tree

N-Trees use NEO at each leaf and therefore majority game consistency is achieved at the leaf level. However, at various times throughout the game, the scope of updates may exceed the boundary of the leaf and the event will be propagated through the N-Tree by the event propagation method described previously. The problem then arises that we must maintain consistency across NEO groups of players at the leaf level.

To address this problem, we use a similar voting mechanism to NEO: an update is only accepted when a majority of players accept the update. When an update exceeds the scope of the leaf it is in, the update is propagated to other leaves whose boundaries intersect with the scope of the update. Like NEO, we first send the digitally signed hash of the update for other players to vote on. When the hash reaches a new leaf, all the players in the NEO group of the leaf vote on the hash. The leader of the leaf then takes a count and forwards this tally up the leaf to the root of the tree. Each intermediate node along the path to the root tallies the votes from its children and forwards the results up to the root. Once the final tally reaches the root, the root of the N-Tree sends the result back to the leaves. Figure 4.3 demonstrates tallying the votes. At this point, the originator of the update will receive the update and decide by the vote tally whether or not to send the plain-text update.

Note that we do not need to use the actual root of the N-Tree to tally votes for all updates with scopes that exceed the boundaries of their leaves. Instead, each node in the N-Tree can act as the root of the update if their scope encompasses the entire scope of the update. This prevents the updates from taking an unnecessarily long time to be voted on since the votes will only need to be propagated up to the node that encompasses the event.

4.4.5 Bootstrapping

To bootstrap and help maintain N-Trees, we use an underlying DHT. In particular, we use CAN [11], though N-Trees can be mapped to any DHT such as Chord or Pastry [54, 12],

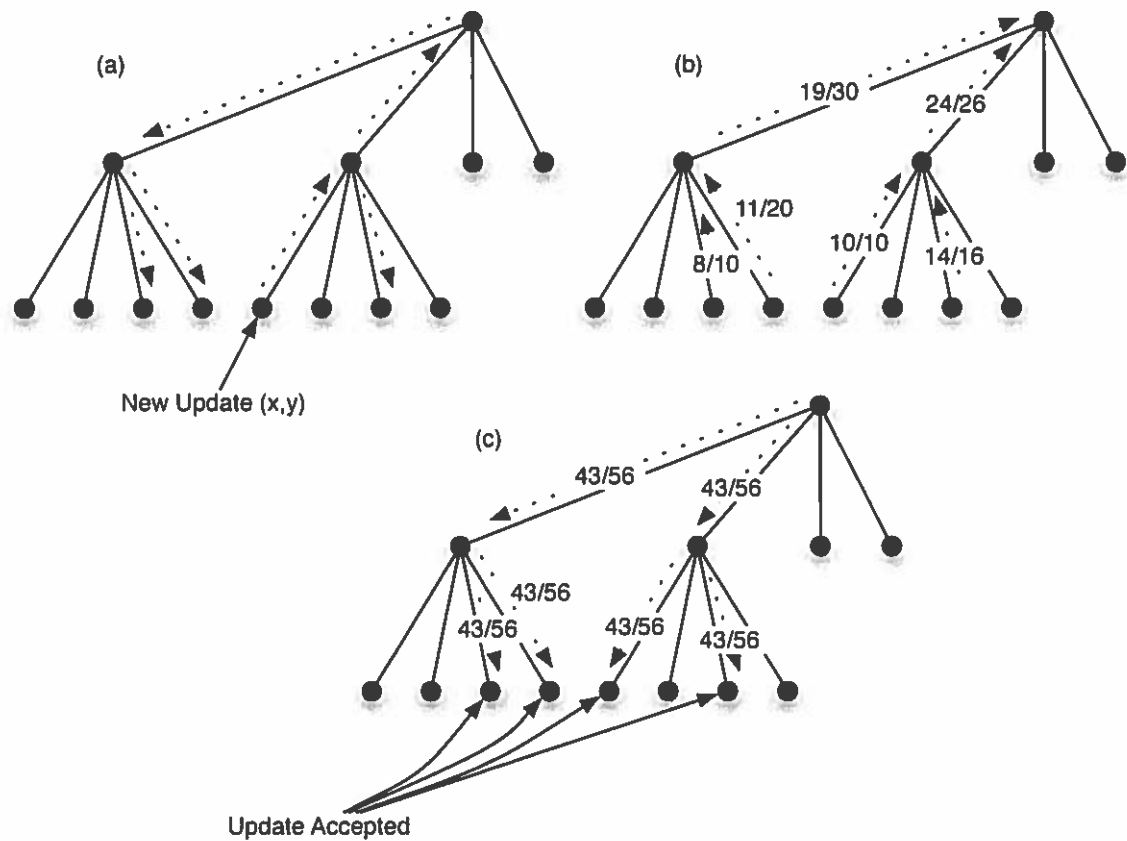


FIGURE 4.3: *N-Tree Voting:* In (a), a new update is generated at (x, y) and its hash is propagated to the leaves with which it intersects (the dotted arrows indicate the direction of travel). In (b), each NEO group at a leaf votes on whether to accept or reject the hash of the update. This vote is sent to the parent node and each parent sums the accept and reject votes, and further sends the vote up to the parent until the root is reached. In (c), the root has received all votes and sends the result down to all affected leaves. Once the vote reaches the leaves, it is accepted and the originating player can send the plain-text update to the appropriate leaves.

and possibly unstructured peer-to-peer networks. DHTs have already been shown to be scalable, but they are optimized for data lookup based on a key. Thus, they provide a mapping of keys to nodes in the DHT.

CAN is an N-dimensional torus and subdivides the key space when the load of a node exceeds some threshold. Multi-dimensional virtual worlds map naturally onto CAN's keyspace because we can configure CAN to have multiple dimensions with a one-to-one correspondence between a coordinate in the virtual world and a coordinate in CAN. Figure 4.4 demonstrates mapping (x,y) in the virtual world to the CAN keyspace.

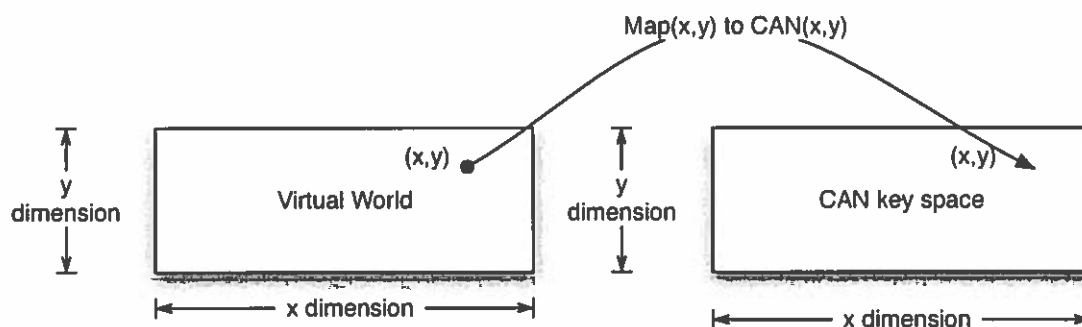


FIGURE 4.4: Mapping virtual world coordinates into CAN: Coordinates in the virtual world simply map directly to coordinates in the N-dimensional torus of CAN. In this case, the (x,y) coordinate in the 2D world maps to the (x,y) coordinate in CAN's key space.

Peers in the N-Tree use the DHT to register their IP addresses with locations in the virtual world. When a new node wishes to join the system, they use the DHT to route a message to the coordinate they belong to. This message will be routed to the system in the DHT in charge of that key, which will have the IP addresses of players within that portion of the key space.

The advantage of the DHT is that if some node in the DHT has failed and entries are missing, the next responsible node in the DHT will respond with the entries it has. Instead of having every node join at the root of the N-Tree and causing the root of the N-Tree to handle the burden of all joins in the game, joining is distributed among the tree by whatever distribution in the virtual world that players exhibit. Thus, as long as a single node in the N-Tree can be discovered on the DHT, new peers can join the system.

4.4.6 Joining the N-Tree

To join the N-Tree, a peer queries the DHT using her virtual world coordinates as the key for some set of nodes in her location of the application space. She then sends a *join* message to one of the nodes that includes the peer's current scope in the application space. The receiving node looks at the scope of the join and determines whether to forward the message up or down the tree. Eventually, the *join* message reaches a node that is within the peer's event scope, and this node notifies the peer so that it can join the N-Tree at this location.

Each node has the ability to divide its scope based on the communication needs of the application. The reason that subdivision is application specific is that some applications have a much higher event rate than others. Applications with a high event rate will need to subdivide the state space to a high degree so that the t -way communication between the t peers in a leaf is reduced as much as possible without hurting the performance (or security) of the application.

4.4.7 Leaving the N-Tree

To leave the N-Tree gracefully, a peer sends a *leave* message to all other peers in its node (or *group*). A new leader is elected and the parent node is notified if the departing peer is the node's current leader. For an ungraceful leave, either a parent or a group will notice that a peer is missing when they try to forward an event to it. The case where the peer is not a leader is trivial. However, when the peer is a leader, the protocol must handle rejoining the group to the tree. If the group noticed that the leader is missing before the parent did, then they can simply re-elect a new leader and notify their parent.

On the other hand, if the tree is attempting to forward an event to a subtree, then it may discover that a child is not responding to an event. To handle this case, leaders should periodically send membership lists to their parent so that the parent can quickly send the event to another group member. Furthermore, the parent can queue updates until the subtree is reconnected. However, even if the membership list of the parent is outdated, the parent can locate members of the subtree through the DHT.

If a peer leaves that is both a leader of a group and of one or more parent nodes, then the group will first elect a new leader and then the leader will contact other leaders to elect higher level node leaders. The DHT allows peers to discover other leaders if they do not already know of them.

4.4.8 Subdivision of Leaves and Collapsing of Branches

N-Trees are configured with two thresholds, t_s and t_c , which are used to determine when a leaf is subdivided or when the children of a node are collapsed into a leaf respectively. To determine when to subdivide a leaf, the leader of a NEO group monitors the population in comparison to t_s . When the population exceeds t_s , the leaf is evenly divided along all N dimensions, players are grouped according to their location the new subspaces, and leaders are elected for each subspace. Each new subspace forms a new NEO group.

Nodes also monitor the population of their children. If the collective population of the children falls below t_c , the node informs the branches that they will be collapsed into a single leaf with the node as the leader. Once the node becomes a leaf, and therefore a single NEO group, a new election can be held to elect a new leader for the leaf.

4.5 N-Trees Consistency

Now that we have extended event ordering beyond NEO to include events between leaves in the N-Tree, we must ensure that the N-Trees maintain consistency. We extend the notion of majority game consistency to include scoped updates with each NEO group:

Definition 4.5.1. Majority★ Game Consistency: The result of any game is the same as if the updates from all players were executed in some sequential order, and only the updates seen by a majority of players and within the scope of that majority appear in this sequence in the order specified by the originating player.

The distinguishing feature of Majority★ Game Consistency and Majority Game Consistency is that only those updates voted on by a majority of players *within the scope of the update* appear in a sequential ordering of the updates. Our prior definition of majority game consistency assumed that all players experienced all updates.

Figure 4.5 illustrates two different rounds in game time with two NEO groups in an N-Tree. In Figure 4.5a, the shaded areas indicate which players accepted Update A at time r in NEO group I and II respectively. In Figure 4.5b, a different set of players from NEO groups I and II accept Update B at time r' . However, at the end of the game, all players will have agreed that Update A and Update B were part of the sequence of events. The players who are not in the shaded majority will need to recover the updates and insert them in their sequence of events.

As with NEO, consistency in N-Trees only applies to the ordering of updates within a game and does not necessarily correspond to human notions of consistency in the virtual world. In other words, rollback may cause the game to appear inconsistent to a player since we cannot really take back the moves, from a human perspective, that occurred prior to the rollback. However, N-Trees is consistent if at the end of the game all players have accepted the same set of updates at each round if they are in the scope of those updates.

N-Trees also have an event horizon, like NEO. We define the N-Tree event horizon to be:

Definition 4.5.2. N-Tree Event Horizon: The minimum event horizon of all NEO group event horizons at the leaves of the N-Tree.

The event horizon is important for very practical reasons in an MMOG. The event horizon allows us to determine when the earliest time (round) for which the state of the game has been agreed on by all players. In an MMOG, which has persistent state, we know we can commit state that is prior to the event horizon since it can no longer change due to rollback. Thus, buffers can be cleared and we can commit to the game database once the event horizon has passed.

Theorem 4.5.3. N-Trees provide majority★ game consistency.

Proof. By Theorem 3.3.11, we know that all players in a NEO group achieve majority game consistency for locally scoped events. Therefore, we only consider updates with scopes that exceed the boundary of a leaf.

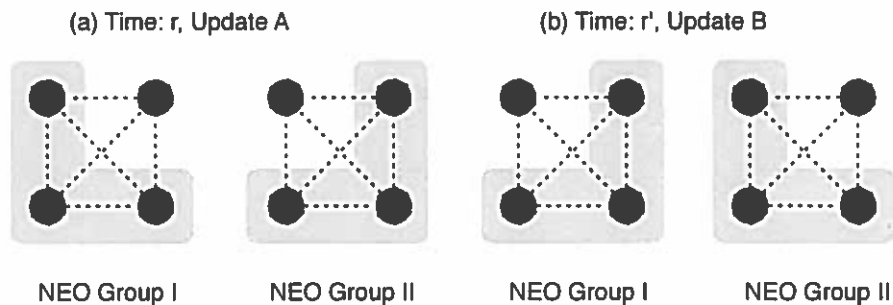


FIGURE 4.5: *N-Tree consistency:* Updates which affect more than one NEO group must be accepted by a majority of players from those NEO groups. In this figure, the shaded area indicates the majority of players which accepted the Update in each NEO group. In (a), at time r , both NEO groups I and II accept Update A, while at time $r + 1$, both NEO groups accept Update B, albeit with a different set of players. At the end of the game, all players will have agreed on the same set of updates, where in this case it includes Updates A and B. Those players not in the majority will have to recover the update and most likely rollback.

The primary question we must answer is whether or not an update can be accepted by some player s while being rejected by other players. This can only happen in two cases:

- (i) Some set of players do not receive the update even though it is in their scope.
- (ii) Some set of players believe that an update was accepted when the majority voted to reject it, or they believe that an update was rejected when the majority voted to accept it.

For case (i), recall that N-Trees use reliable transport between nodes in the N-Tree. Therefore, we can assume that votes and updates can reach their intended nodes, unless the node has failed. In that case, the N-Tree protocol will replace the node and the update or vote tally can be resent or routed through the underlying DHT. Thus, we can guarantee that players will receive the update if it is within their scope.

For case (ii), assume that players do not lie about voting tallies (we address security and the case where players lie in Section 4.6). Since we know that voting tallies are guaranteed to reach their intended nodes, all players within the scope of the update will receive the correct tallies. Therefore, all players will agree on the update. \square

Safety and Liveness of N-Trees

We now show the *safety* and *liveness* properties of N-Trees. The error condition that the safety of N-Trees protects against is that the game state will become inconsistent. The liveness property of N-Trees ensures that N-Trees will not stall or halt during execution. This property is important because we need to ensure that event ordering will progress until the game is complete.

Lemma 4.5.4. *Prior to the N-Trees event horizon, N-Trees are safe; thus, no error condition will arise in the execution of the N-Tree protocol.*

Proof. By Theorem 4.5.3, we know that N-Trees provide majority★ game consistency. Thus, prior to the N-Tree event horizon, N-Trees are safe. □

Lemma 4.5.5. *N-Trees are always live; thus the game continues to advance monotonically with time.*

Proof. By Theorem 3.3.7, we know that the NEO event horizon increases monotonically over time. Since the N-Tree event horizon is equal to the minimum NEO event horizon, we know that it also must advance. Thus, the N-Tree event horizon increases monotonically with time and therefore N-Trees are always live. □

Lemma 4.5.6. *N-Trees are safe at the end of the game; thus, no error condition arises when the protocol completes.*

Proof. By Lemma 4.5.5, we know that the N-Trees are live, thus the protocol will continue until the game ends. By Lemma 4.5.4, we know that N-Trees are safe prior to the N-Tree event horizon. Thus, N-Trees will be safe when the protocol completes. □

4.6 N-Trees Security

One important issue with N-Trees is that players who control nodes in the N-Tree have significantly more power in terms of cheating than with NEO. Under the NEO protocol, a player can only alter her packets, but with N-Trees, a player is now able to drop other

players' packets and lie about voting tallies. For example, the leader of a NEO group in the leaf of a tree may simply not forward an update to other players or may say that no one else voted to accept the update.

To address these problems, we propose using multiple N-Trees, similar to the method of using multiple *realities* with CAN [11] and other DHTs. Multiple N-Trees with different nodes acting as leaders can prevent a single player from controlling the root node, for example, since more than one path exists to the leaves of the N-Tree.

The following protocol level cheats are addressed by N-Trees:

Timestamp Cheat

With the timestamp cheat, a player lies about when an update occurred so that it appears the update was simply delayed in the network. While NEO prevents the timestamp cheat at the leaf level, when we propagate updates through the the N-Tree, a player may see this update and immediately issue a competing update timestamped at the same time. N-Trees use a voting mechanism similar to NEO, except that nodes tally votes as they are propagated from the leaves of the tree to the root. Thus, N-Trees can prevent the timestamp cheat since players will vote on accepting a move before seeing it.

Suppressed-Update Cheat

The suppressed update cheat does not work with N-Trees for the simple reason that updates are not accepted if they are not forecast sufficiently into the future. Thus, the voting mechanism will prevent the suppressed update cheat from being viable.

Fixed-Delay Cheat

As with the suppressed update cheat, the fixed-delay cheat does not work with N-Trees because updates that are not forecast sufficiently into the future are not accepted by players within the scope of the event.

Inconsistency Cheat

The inconsistency cheat in N-Trees is not possible because the N-Tree acts as a multicast tree and events are replicated by other nodes. A player cannot send two different updates to two different nodes in the N-Tree.

Collusion Cheat

In the collusion cheat, several players agree on some action that gives them an unfair advantage in the game. The main action that colluding players can accomplish with the N-Tree protocol is the modification of voting tallies. With N-Trees, two colluding players must be controlling the same node in the N-Tree at which point they can lie about voting tallies.

To prevent this cheat, we reassign node leaders in the N-Tree frequently. While this does not guarantee that we can prevent collusion, we can reduce the probability of to a fairly low value. For example, if we maintained 2 copies of the N-Tree and randomly assigned players to control the nodes, the odds that 2 players were colluding and controlling the same node would be $1/4^h * 1/4^h = 1/4^{2h}$. Given an N-Tree with a height of 10, the probability would be 9×10^{-13} . Collusion prevention methods are an area of future research.

4.7 Analysis of N-Trees

Our analysis of N-Trees assesses the ability of of N-Trees to meet the requirements for scalability and real-time responsiveness under a range of network conditions. We begin with an asymptotic analysis of the cost of joining and leaving the N-Tree, moving to a new leaf, collapsing and subdividing leaves, and event propagation in terms of the number of messages that must be sent over the N-Tree. We follow our asymptotic analysis with an actual measurement study of a real MMOG and use the model derived from it in our simulation-based analysis of N-Trees in Section 4.9.

4.7.1 Asymptotic Analysis

In order to understand N-Trees as a communication structure for scoped events, we analyze their performance in terms of messaging. In general, all operations take at most logarithmic time in terms of the number of peers, while some only take constant time. Table 4.1 summarizes these results. In the following discussion, n is the number of nodes in the N-Tree, p is the number of peers, t_s is the threshold for subdividing a node, d is the number of leaves per branch (i.e., $d = 2^N$ in the N-tree), and h refers to the height of the tree.

As with binary trees, many operations on the N-Tree are based on its height, h . N-Trees do not try to balance themselves to achieve logarithmic height with respect to the number of nodes in the tree. Thus, in a worse case scenario, the height of the tree could be $O(p/t_s)$. However, this scenario only represents the case when all players are in the same location in the virtual world (a case we assume to be extremely rare or non-existent as p gets sufficiently large).

Unfortunately, no prior scientific measurements have been conducted that show the distribution of players in the virtual world, though we address this shortcoming with an initial population study in Section 4.8 Anecdotaly, game designers desire a uniform distribution of players in the virtual world to keep the level of inter-player messaging low (so that the game can scale to a large number of players). Given a uniform distribution, the N-Tree is balanced and $h = O(\log_d n)$. Recall that the height of the N-Tree is based on the number of *nodes* in the tree, not the number of *peers* in the game. Each leaf in the N-Tree represents a NEO group where the threshold t_s is the maximum number of peers in a NEO group. Assuming a uniform distribution, we have $n = O(p/t_s)$ since each leaf is subdivided when the number of peers in it exceeds the threshold value t_s .

Joining is a $O(\lg p) + O(h)$ operation. The $O(\lg p)$ timing comes from the search time for most DHTs (assuming all peers are registered in the DHT). Once a node is found for the N-tree, a peer will most likely be able to join the N-Tree in constant time. However, if the DHT is not up to date, it can take an additional $O(h)$ time to locate an appropriate node through the N-Tree.

TABLE 4.1: *Asymptotic messaging costs:* p is number of peers, h is height, n number of nodes, $d = 2^N$ (or the number of leaves per branch) in the N-Tree.

Operation	Player distribution	
	Pathological	Uniform
New Member Join	$O(\lg p) + O(h)$	$O(\lg p)$
Move to new node	$O(h)$	$O(\log_d n)$
Amortized movement	$O(1)$	$O(1)$
Leave	$O(1)$	$O(1)$
Collapsing branch	$O(1)$	$O(1)$
Subdividing leaf	$O(1)$	$O(1)$
Event propagation	$O(h)$	$O(\log_d n)$

Once a node has joined the N-Tree is joined for the first time, moving in the N-Tree is a much faster operation. The majority of player movement is local in the virtual world, so a player simply leaves their node and joins a neighboring node, requiring at most $O(h)$ time. However, $O(h)$ is the worse case scenario where we have to move between two branches of the root of the tree. Note that we can mask the latency required to join a new node by joining the node early. In other words, the peer is temporarily the member of two neighboring nodes and leaves the originating node once she has fully transitioned to the neighbor. For global movement, e.g. warping from one part of the world to another, the player leaves and rejoins the N-Tree at their destination as described above, similar to joining the N-Tree for the first time.

Leaving, on the other hand, is a simple $O(t_s)$ operation, where t_s is the maximum number of members in a leaf (the application threshold before subdivision). When a node leaves, it must contact the other $t_s - 1$ members to notify them that it is leaving.

To analyze the complexity of subdivision, we simply examine our leader election protocols. A constant number of messages are required to initiate subdivision, i.e. $O(t_s)$. Our leader election protocol requires $O(t_s^2)$ messages since all peers in the leaf must exchange messages with each other.

Subdivision has another cost in the amount of state that must be stored at each node. For N-dimensions, each node must store 2^N pointers to children. Applications designers should be motivated to reduce the application state space when possible to avoid a state

explosion. We believe that most applications will not have a large number of dimensions in the application state space. In particular, most games use 2 or 3 dimensions and therefore $N = 2$ or $N = 3$ with our N-Tree.

N-Tree collapsing is a $O(1)$ operation. We assume in this case that $O(2^N) = O(1)$ because the N value is a constant value set by the designers of the application. For peer-to-peer games, N would probably be 2 or 3, meaning 4 or 8 messages at most for collapsing a branch. Each node is periodically sending membership lists to its parent. When the parent notices that the number of peers in its subtree is less than or equal to t_c , the parent sends out a message to the peers and collapses the tree. To prevent repeated subdivision and collapsing, the minimum and maximum threshold for a leaf should be different.

Event propagation is a $O(h)$ operation, where h is the maximum height in the tree. In the worse case, a leaf at the lowest level of the N-Tree generates a global event that must travel from the leaf up to the root of the tree and back to all NEO groups at the leaves. As with joining, the worse case scenario is $h = O(p/t)$, which again represents a case where most players are in a single location in the game. With a uniform distribution, $h = O(\log_d n)$. However, even if most players are located in a single location, empirical observations indicate that the majority of the events are local so that most events will not need to traverse the entire N-Tree. In most cases, events will travel through only a few levels of the tree, keeping the cost of event propagation low.

4.7.2 Performance Analysis with Local Event Propagation

In addition to looking at the asymptotic performance of N-Trees, we wish to see how the performance of N-Trees affects the latency players would experience if N-Trees were used to propagate events, especially in comparison with similar architectures. We performed a preliminary study to compare N-Trees to the only two existing studies.

In the architecture by Knutsson et al. which we call multicast with regions (MCast Regions), experiments were run using their simulator with 1000 and 4000 players over a virtual world divided into 100 and 400 regions, with players uniformly distributed over the world [6]. Players were connected over a randomly generated topology with latencies between 3 to 100ms. For Mercury, the authors only simulated 20 and 40 players in each

simulation, but uniformly distributed players and assumed a random way-point model of movement for players [4]. In the Mercury simulations, they assumed players were connected with a star topology, each with a 20ms delay between each other and the simulation field was about $2n$ times the maximum distance a player could move, for n players. Neither work considered events that were not local to the players or a particular object.

To analyze the performance of N-Trees in a similar setting, we assume a uniform distribution of players, only local events, and a 20ms delay between players, as with the Mercury simulations. We then extrapolate the data from [6] and [4] to get an idea of how N-Trees, Mcast Regions and Mercury compare for the dissemination of messages.

Using this data, we plot our hypothesized performance based on N-Trees and using the simulation parameters common to Mercury and Multicast regions using Scribe. Please note that the graph is purely speculative, but demonstrates the effectiveness of organizing players by their application-level interest instead of by the shortest-path multicast tree. Figure 4.6 shows our hypothesis graphically.

We believe our hypothesis will hold for the simulation parameters used in [4, 6]. Our reasoning is if players are uniformly distributed and events are only local (i.e., they do not propagate through the N-Tree), then N-Trees perform optimally. The movement models used in simulations by Bharambe et al. and Knutsson et al. ensure that the population will stay uniformly distributed. Thus, the N-Tree will be perfectly balanced and subdivided so that players close together exchange events directly with each other. With a 20ms delay between all players, this results in an average 20ms delay, regardless of the population of the game.

Clearly, a more complicated and realistic set of experiments are needed to validate N-Trees. Indeed, the true test for N-Trees, Mercury, and Multicast regions, occurs when players have a non-uniform distribution and events have scopes that occupy more than a single point in space. Knutsson et al. suggest dynamic region adjustment for these situations, but do not describe how to accomplish this. Mercury, on the other hand, with its query language should be able to handle *crowded* situations more gracefully. As such, we expand our experimental analysis of N-Trees in Section 4.9 to include a power-law distribution of players.

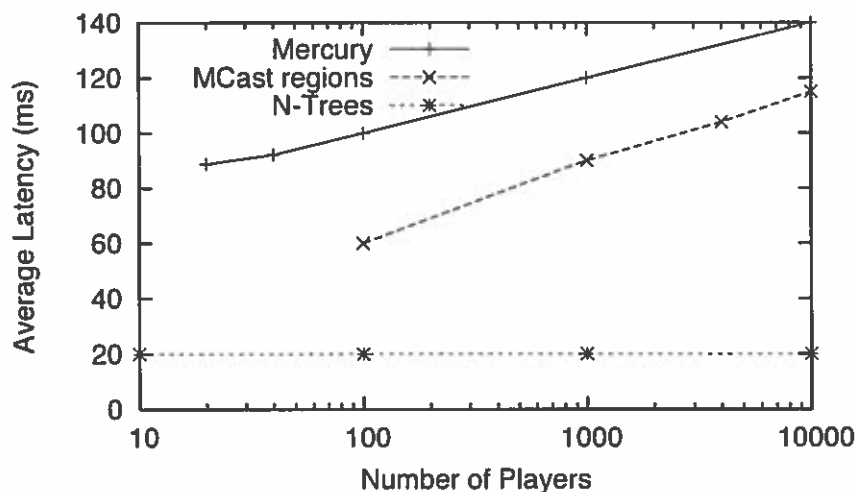


FIGURE 4.6: Latency for local events with N-Trees, MCast Regions, and Mercury: For Mercury, the first two data points at 20 and 40 players are taken from [4], with the rest of the points derived from their measured number of hops in the Mercury DHT. For Multicast regions, points 1000 and 4000 are taken from their simulations in [6], with the rest of the points extrapolated from their data and Scribe performance. N-Tree performance is hypothesized based on the simulation parameters of previous work in [4, 6]

4.8 Measurements of Virtual Populations in MMOGs

In order to perform a deeper and more realistic evaluation of N-Trees, we develop a model of player movement and behavior over time to use with our simulations. In particular, we have performed a measurement study of virtual populations on the popular MMOG World of Warcraft and we have developed a player movement model to drive our simulations of N-Trees in Section 4.9.

While previous research has addressed network characteristics, such as where players are located geographically and inter-packet arrival times [61, 74], these studies do not address the behaviors of players online and therefore cannot be used independently to create an accurate simulation model for MMOGs. Our study is, to the best of our knowledge, the first measurement study of virtual populations and their behavior in commercial MMOGs.

To create a viable model for MMOGs, five aspects of the virtual populations from MMOGs must be measured:

- *Population changes over time:* Population changes over time are important because we need to understand the impact on resource provisioning. Resource provisioning is affected by aspects of the population such as cyclical peak demand patterns and the magnitude of the difference between the peak and lowest population sizes during a cycle. We hypothesize that populations exhibit a prime-time diurnal pattern, mimicking television viewing patterns, because the majority of people that play MMOGs tend to have daily obligations, such as work or school and tend to play in the evenings.
- *Arrival rates and session durations:* The arrival rate and session durations of players are necessary to measure because a fast arrival rate and short session duration indicates that a significant amount of *churn* is occurring. The design and testing of MMOGs need to take this into account. For example, the overhead of overlay management increases in a P2P network as churn increases.
- *Spatial distribution of players over the virtual world:* The distribution of players over the virtual world is an important factor because different distributions can have significant effects on the performance of a given architecture or protocol. Architectures will need to take issues like congestion control and load balancing into consideration if the distribution of players tends to be non-uniform. We hypothesize that player distributions are most likely *not* uniform and are probably close to power-law.
- *Movement of players versus time and space:* The fourth important aspect to understand in modeling MMOGs is where and when do players move around in the virtual world. Player movement can have a huge impact on protocols. Consider the case where players visit many zones and only remain in each zone for a small amount of time during their session. In this case, the amount of player movement can have an adverse affect on the seamless hand-off of players between nodes in a cluster or distributed system.
- *Temporal distribution of events and distribution of scope sizes:* Knowing how frequently events are generated and the distribution of their scope sizes will help us understand how far events are propagated into N-Trees.

We performed an extensive set of probing-based measurements over World of Warcraft [10], one of the most popular MMOGs in North America. World of Warcraft is estimated to have around 6 million subscribers and uses a client/server architecture typical of all commercial MMOGs. The client provides a scripting interface and allows end-users to create *add-ons* for the game. Using this interface, we wrote a set of scripts which measured and recorded populations, distributions of players across the virtual world, session times, and player movements. Due to a limitation of the scripting interface, we were unable to capture event information, such as how frequently they were generated or what scope they had.

Our preliminary results show that populations change according to a prime-time schedule. This validates prior network traffic measurements by Kim et al. [74] and Chambers et al. [61]. Our results show that player distributions and churn occur according to a power-law distribution. Players tend to only visit a small number of zones and remain in each zone for 10s of minutes if they play longer than 15 minutes.

4.8.1 Background

To validate research on distributed architectures for MMOGs, researchers have used simulations with artificial workloads, with the exception of Baughman and Levine in [3], who used real traces from a small networked multiplayer game called XPilot [80]. The artificial workloads used by past researchers were similar and typically consisted of a virtual world evenly divided into zones with a uniform distribution of players across the virtual world. Player movement was simulated by having players remain in their zone for some uniformly distributed amount of time and then randomly choosing a new zone to travel to.

Little work has been done on the characteristics of player populations in MMOGs. Some website have measured population sizes and other statistical game information from their favorite MMOG, including data such as the kind of equipment that players have, where treasures are located, and how often particular items re-spawn. These informal measurements are used by players for game strategy and are not part of a systematic and scientific measurement study.

With respect to scientific measurement studies, Chambers et al. have studied various network conditions related to players and the server of small networked multiplayer games [61]. Similar to our data, their measurements also show diurnal patterns of game populations. This work focused on measuring network traffic patterns, and did not measure characteristics of virtual populations.

Kim et al. measured network patterns on Lineage II, a popular MMOG in Korea [74]. Their work focused on network packet sizes, RTTs, session times and inter-session arrival times. Their data on session times appears to show a similar power-law distribution of playing times, with 50% of all playing sessions lasting less than 26 minutes. Their work focuses on network traffic and does not investigate MMOG virtual populations.

Tan et al. measured player mobility in small networked first-person shooter (FPS) games and designed a mobility model for that class of games which they call the Networked Game Mobility Model (NGMM) [81]. They showed that the typical random waypoint mobility model is not sufficient for modeling player movements when compared to actual traces of player movements from FPS games. Unfortunately, FPS games typically host only 16 to 32 players, with player movement that differs significantly from MMOGs. Thus, the mobility model for FPS games does not extend to MMOGs.

4.8.2 Measurement Technique

In order to measure the virtual populations and behaviors of players in an MMOG, two methods can be used. The first method is to analyze logs generated directly from an MMOG. This method has a clear advantage that it will always be more accurate than other methods. However, game companies are very protective of the proprietary technologies used to run MMOGs and are typically unwilling to share log information which may give competitors insight into their designs.

The second measurement method is to use a probing-based technique to try to infer properties of the system. Because we were unable to convince any of the MMOG game companies to share their logs for scientific study, we use probing-based measurements. We collected data from a highly populated World of Warcraft realm.

We designed a set of scripts that run from the game client. The World of Warcraft client is designed with a Lua [82] scripting interface so that end-users can write add-ons to change the user interface and provide modules that players can use to make the game easier to play or more enjoyable. We use this interface to record information about the virtual populations.

The World of Warcraft server provides an interface into the *who* service and provides a *friends* list, both which assist our measurements. The *who* service is a service that the game provides which lets us query which players are currently online and returns additional information to us such as what zone they are in. For scalability reasons, the *who* service only returns 50 results per query and only accepts one query every few seconds. However, we can provide parameters and simplified search expressions with our query. For example, we can query about only the players in a particular zone or about players with names that begin with a particular set of letters. As such, when we query and receive 50 results, we know that we need to subdivide our current query into a smaller set of players. This technique allows us to systematically search the entire set of players currently online.

The second interface, the *friends* list, allows us to store updated information on 50 players in the game. World of Warcraft sends an event whenever one of our 'friends' logs into or out of the game. In addition, the server provides information about the player such as what zone they are in. Therefore, when World of Warcraft informs us that a friend has left, we can timestamp our data indicating their exact departure time.

The main difference between the two services is that with the *who* service, we can query the server repeatedly to create a snapshot of the current virtual population. Because each measurement takes up to five minutes, the population is changing while we are measuring. Thus, we take back-to-back snapshots of the population which give us an idea of how much fluctuation is occurring while we are measuring. For example, the union of the two measurements represents the stable population over the measurement period.

On the other hand, the players seen in the second snapshot that are not seen in the first snapshot represent the arrival rate over the time it takes to complete the second snapshot. For example, if we see 200 new players in the second snapshot, and the time from starting the first snapshot to finishing the second snapshot is t seconds, then the arrival rate is $200/t$ arrivals per second.

With the friends list, we are able to query the status of our friends more frequently. In our study, we ask for an update every 30 seconds. This allows us to track player movement and session lengths on a more fine-grained scale. However, World of Warcraft still sends us a message immediately when a friend leaves, allowing us to determine their exact session length. In comparison, a single snapshot of the current virtual population takes approximately 5 minutes. During that time, a single player may leave and rejoin the game, and may have moved between several zones.

We initially populate the friends list with 50 randomly selected new arrivals from the back-to-back snapshots produced by our *who* queries, i.e., those players in the second snapshot that do not appear in the first snapshot. When a friend leaves, the server sends an event indicating so, and we add a new friend as soon as the next back-to-back snapshot has occurred. This allows us to track the session lengths of those players on our list with more accuracy and it allows us to monitor which zones they are currently in (and how long they stay there) until they log out of the game.

We measured data from World of Warcraft over several 24-hour periods, over a period of 3 months. World of Warcraft is divided into over 100 realms and we measured from three of the most populated realms. We determined which realms had the highest populations from a web page provided by Blizzard that shows the current status of each realm, including populations in general terms (low, medium, high, max (queued)) [83].

We record the current population every 15 minutes by taking two back-to-back measurements from the *who* service. Due to the desire to not flood the server with constant *who* queries, we limit our measurements of the entire population to 15 minutes. In addition, World of Warcraft is divided into two *factions*, or groups of players. Belonging to one faction prevents you from querying about players in the other faction. Thus, we are required to measure the population from both factions.

Our data collection methods were constrained by the game environment in a number of ways. First, we had to log into the game as a player and use game-based probes to gather our data. Obviously, logs generated by the game directly would have been both easier and more accurate. Second, anti-cheating mechanisms built into the game prevented us from running simple, periodic scripts to completely automate our data collection. These mechanisms automatically log a player out of the game if she does not interact with the

game every 15 minutes through a key press on the keyboard. Thus, we had to run the game in the background and periodically move the character around to keep the measurements going. This limitation severely constrains the ability to measure the game consistently over a long period of time.

As future work, we plan on extending our data collection period for significantly more time, over various realms, and on different MMOGs. However, we feel these initial measurements are an important first step in understanding the characteristics of MMOG populations. In addition, they provide an initial set of parameters to drive our simulation of N-Trees in Section 4.9.

4.8.3 Measurement Results

In our measurements, we were able to study the population changes over time, the arrival rates and session durations, the spatial distributions of players over the virtual world, and the movement of players versus time and space.

Population Sizes over Time

We first measured population sizes over time, to see if the populations fluctuated with respect to the time of day. Our hypothesis was that more players were online during evening hours, due to other daily obligations such as work and school. Figure 4.7 shows a 24-hour set of measurements from one of the most populated World of Warcraft realms. These measurements are the sum of the population from both factions.

Figure 4.7 shows a typical 24 hour day, which has random fluctuations hour to hour. One difficulty in with our tools is that we are unable to measure the game for long periods of time. World of Warcraft is designed so that a player must physically press keys on the keyboard periodically or they will be logged out of the game. This *feature* is designed to prevent players from writing programs that run their characters automatically. Thus, measurements require someone to be physically present to keep the game running.

In Figure 4.7, the X-axis is the hour on a 24-hour Pacific Time Zone clock, but the server measured here is actually located in the Central Time Zone. Note that players do not choose servers based on time zones, but on the more geographically broad category of

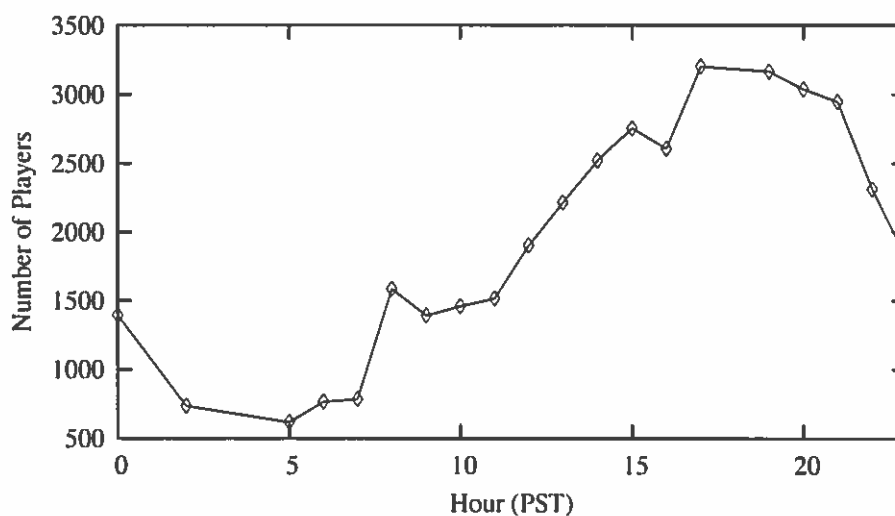


FIGURE 4.7: *Virtual population over time:* As expected, the population increases during *prime-time* hours (5PM PST - 9PM PST). Note that the population has a 5-fold increase at 5PM from its lowest census mark at 5AM. This result is similar to network measurement studies on multi-player games which also show cyclical patterns associated with prime-time peak populations.

continent. Thus, players in the United States can choose to play on any of the US servers, while European players play on European servers.

We note two important aspects of these graphs. First, one might be surprised that the population peaks at close to 3500 players. In fact, if a player tries to log into the game during that peak period, they will be queued and have to wait before entering the game (we determined this by trying to log into the game with a separate client during the peak period). We speculate that 3500 might be the actual ceiling on the number of concurrent players on the realms we measured and that due to churn, we are unable to record all 3500 players. Finally, this number shows why MMOGs need hundreds of realms to support millions of subscribers and is consistent with values reported on Everquest [1].

The second important aspect of this graph is that we see an almost 5-fold increase in the number of players from the lowest point (at 5AM PST) to the highest point (17PM PST). This means that servers must be over-provisioned to handle peak loads during the evenings and are only partially loaded during the early mornings.

Session Durations

The second goal in our measurements is to understand the duration that people play the MMOG. Empirical evidence would suggest that players tend to log on and play for a long period of time. Anecdotal stories talk about players who play for days on end. However, our measurements in Figure 4.9 show that this is not entirely true.

Figure 4.9 shows the measured time playing versus the rank of the duration of play—that is, each session is ranked from the most number of minutes played, to the least. 15 minutes is the shortest play duration that we can measure, which is seen in the graph as a horizontal bar from about rank 3,000 to 10,000. Thus, approximately 70% of the time the playing sessions were less than 15 minutes. To see this more clearly, we plot the CDF of the session duration in Figure 4.8.

These results show that MMOGs have a fairly high amount of churn about every 15 minutes. Because we only measure every 15 minutes, this may be a conservative estimate. Our results show that a large number of people log on and log off over a short period of time. We believe that what may be happening here is that players may be logging on to check to see if friends or guild members are currently online, and if not they log off. If this is true, then the implication is that load on an architecture could be reduced by providing an external *Friends* list that doesn't require logging into the game.

However, to get a clearer picture of the session times, we examined the data collected on session lengths from our *friends* list. This data allows us to track exactly when we first see a player to when the player logs off. Figure 4.10 shows the session durations based off of our friends list with the CDF in Figure 4.11. This data shows us that players continue to play between 1 minute to over 600 minutes (10 hours). However, the CDF shows us that over half of the players play for less than 1 hour in a single session (though they may log in multiple times during the day).

Arrival Rate

We measured the arrival rate, i.e. new sessions, in order to understand how many arrivals we are getting per hour. As expected, the arrival rate depends on the hour of the day, with prime-time hours having a larger arrival rate. Figure 4.12 shows our measurement.

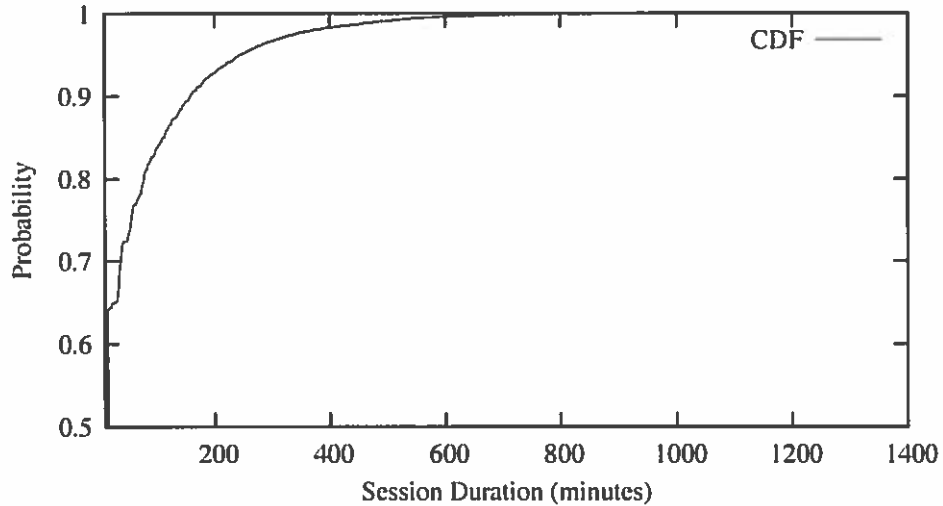


FIGURE 4.8: *CDF of play duration:* This CDF clarifies the distribution in Figure 4.9. The left part of the CDF curve hits 0 probability at 15 minutes, because we only measure every 15 minutes. 65% of the players have a duration of 15 minutes or less. 90% play for less than 200 minutes (approx. 3 hours).

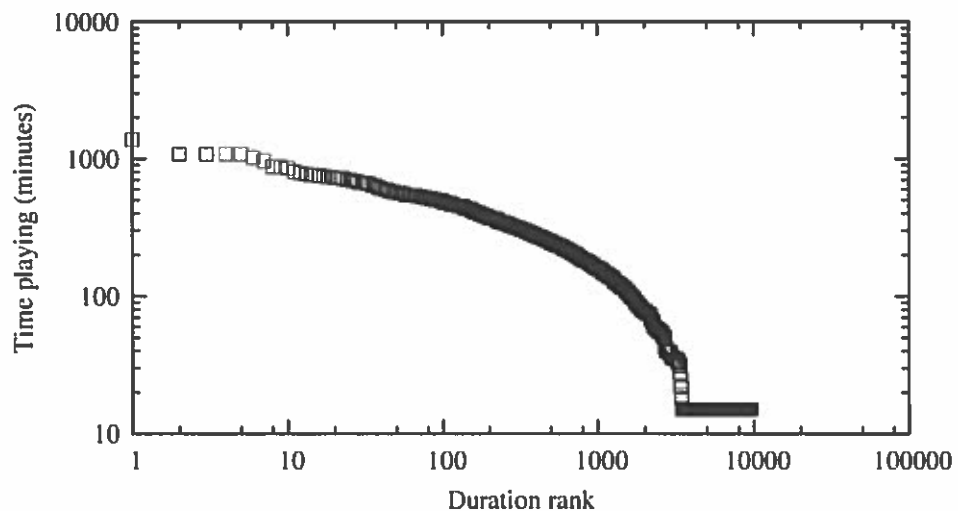


FIGURE 4.9: *Time versus playing duration:* This measurement shows the ranks of the duration of play for each time a player logs in and plays. Most players log in periodically for some time less than 15 minutes (about 70%), while around 10% of the time, players log in for more than 1 hour. 1% of the time, players are playing for more than 12 hours. Overall, the distribution appears to have a power-law relationship.

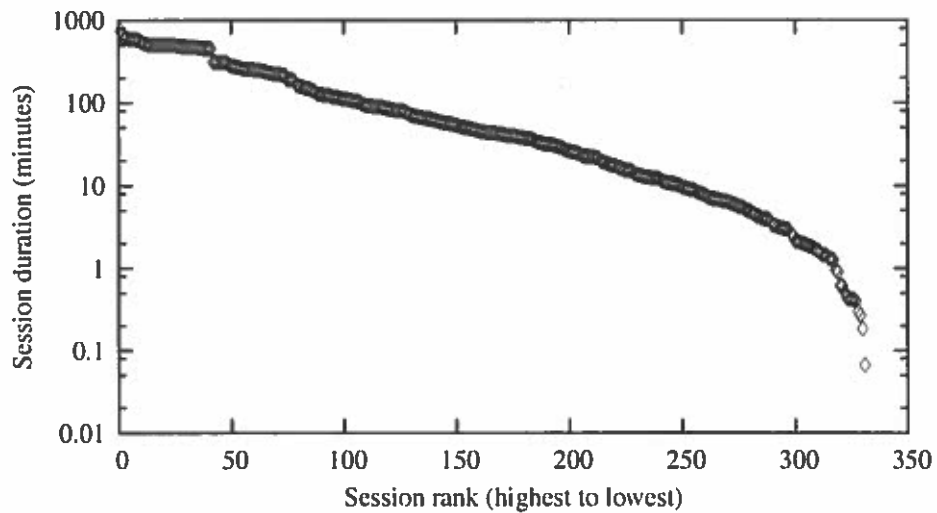


FIGURE 4.10: *Session durations based on friends list:* In this measurement, we used our *friends list* to more accurately capture session times.

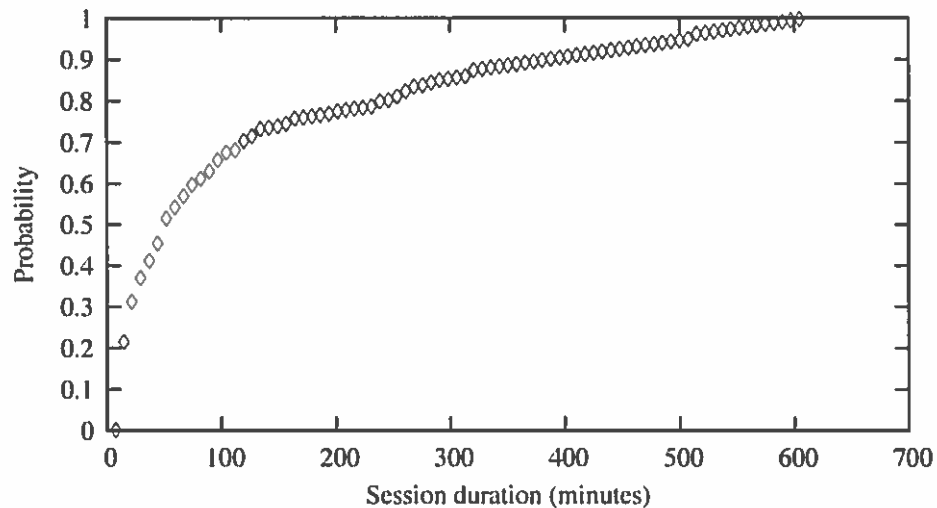


FIGURE 4.11: *CDF of session durations based on friends list:* The CDF of the measurements from Figure 4.10. This CDF illustrates that virtually all measured sessions are less than 10 hours and over 50% of them are more than 1 hour.

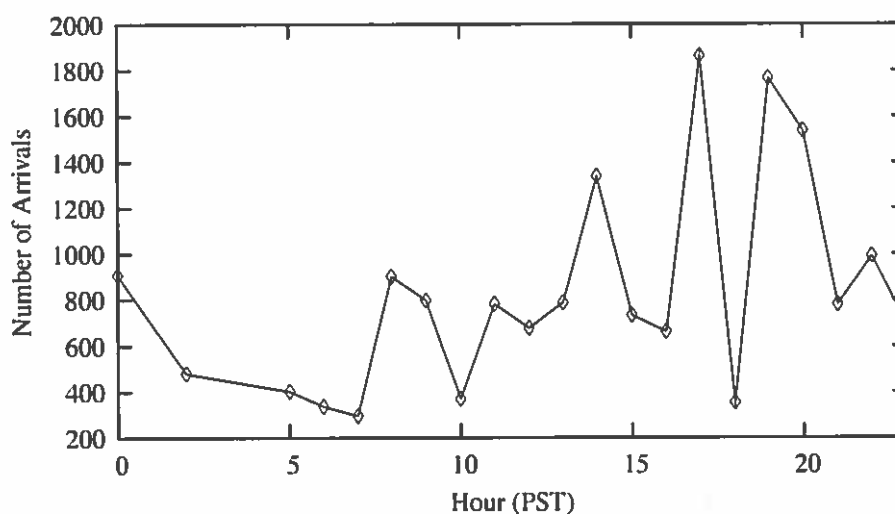


FIGURE 4.12: *Arrival rate over time:* The arrival rate varies, depending on the hour of the day, with prime time hours having a higher arrival rate. The dip at 6pm PST may indicate that the server was at capacity and denying arrivals, while at 8pm the players who were logged on at 5pm were, with high probability from Figure 4.8, logging out so the server had the capacity for new players resulting in the spike in arrivals.

The arrival rate appears to follow the same cyclical pattern that population sizes follow. The minimum arrival rate, at 7AM was approximately 300 new sessions in the hour, versus the maximum arrival rate at 5PM of 1900 new sessions. Note that at 3PM and 4PM the arrival rate fell off, which probably accounted for limits on the servers ability to accept so many new connections at 5PM.

While Figure 4.12 measures the number of arrivals over one day, we also examined the arrival rates seen by our back-to-back snapshots. To determine the arrival rate in this case, we marked when we ended the first snapshot at time t and when we ended the second snapshot at t' . We then calculated the number of new arrivals by counting how many new players appeared in the second snapshot that were not in the first snapshot, which we call c . We can then calculate the arrival rate by $c/(t' - t)$. Figure 4.12 plots the arrival rate as seen in Figure 4.13. The arrival rate varies from less than 1 player per minute to over 32 players per minute. As future work, we plan to correlate the arrival rate with the time the measurements were taken.

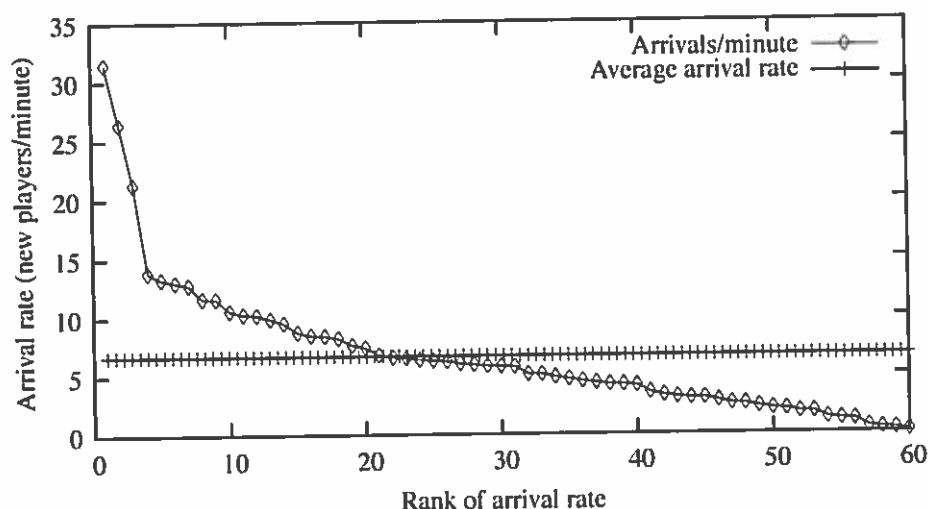


FIGURE 4.13: *Arrival rate of players per minute:* The arrival rate measured from our back-to-back measurement varied from approximately 32 players per minute to less than one per minute, with an average of approximately 7 players per minute.

Spatial Distribution of Players

Our next set of measurements show the distribution of players in the virtual world. Recall that the world is statically divided into segments, called *zones*. We measure how many players are in each zone over a 24-hour period at 15 minute intervals. Figure 4.14 shows the number of players versus the rank of each zone from the most populated to the least and indicates a power-law relationship may exist.

These results indicate a non-uniform distribution of players as we predicted. Indeed, approximately 30% of the zones have fewer than 10 people in them, while about 10% of them have more than 80. Thus, the relationship between the number of players and the zones appears to be power-law.

Our measurements in Figure 4.14 were chosen from the peak period from 5PM to 9PM. Results plotted for other periods during the 24 hour day exhibited the same general pattern, although the magnitudes were smaller due to the lower populations during non-peak periods. Thus, all sets of measurements that we looked at appear to also have a power-law relationship, though we leave the actual curve-fitting to future work.

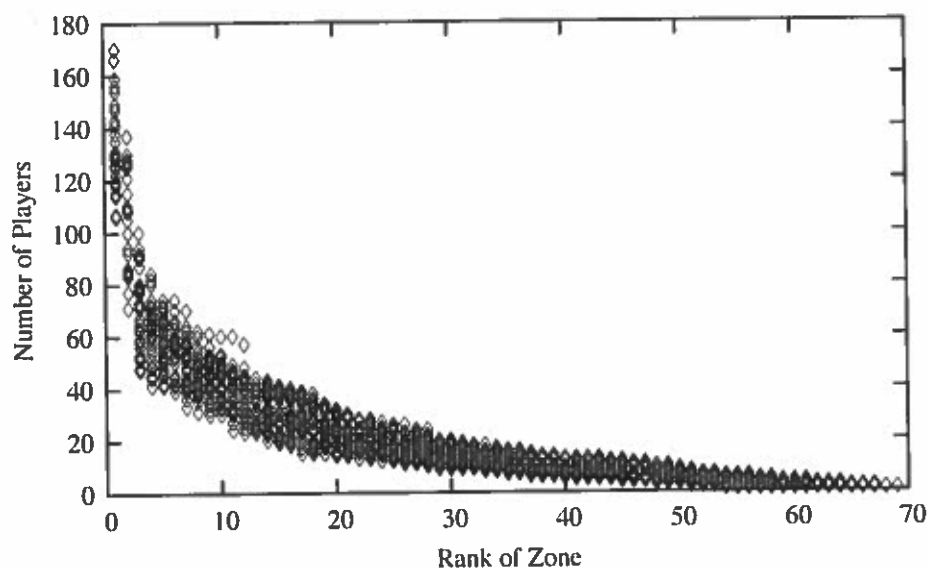


FIGURE 4.14: *Distribution of players per zone:* Zones are ranked from most populated to least populated, with all measurements from the 24 hour period with more than 100 players in the highest ranked zone.

To clarify the results, we plotted the measured CDF of the zone populations in Figure 4.15. The CDF shows that approximately 60% of the zones have fewer than 10 people in them, while a few zones have more than 100 people in them.

Player Movement

The last aspect we measured was player movement. We want to investigate how many zones players visit during their playing time, and how long do they remain in any given zone. To understand the data we measured, we plotted histograms.

Figure 4.16 is the histogram of the percentage of players versus the number of unique zones visited during a playing session. The results show that over 30% only visit 1 zone, while only 15% visit more than 6 zones. This result is expected because zones are tailored for players of a given level. In essence, low level players can only visit a handful of zones, while high level players rarely visit low level zones because they no longer offer a challenge. Furthermore, the previous results that players only play for 15 minutes or less corroborates the measurements that most players only visit only 1 or 2 zones.

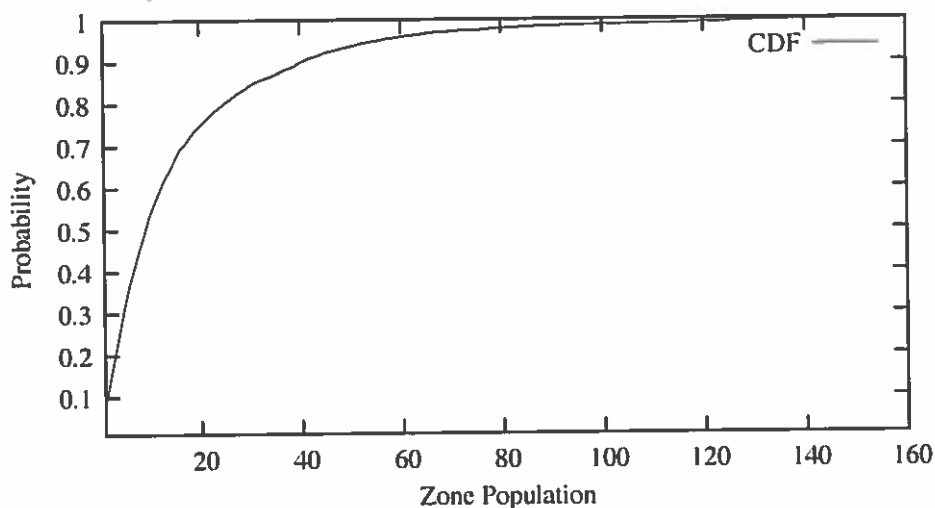


FIGURE 4.15: *CDF of player distribution in zones:* This CDF shows that approximately 60% of the zones have fewer than 10 people in them, while only a few have more than 100 people in them.

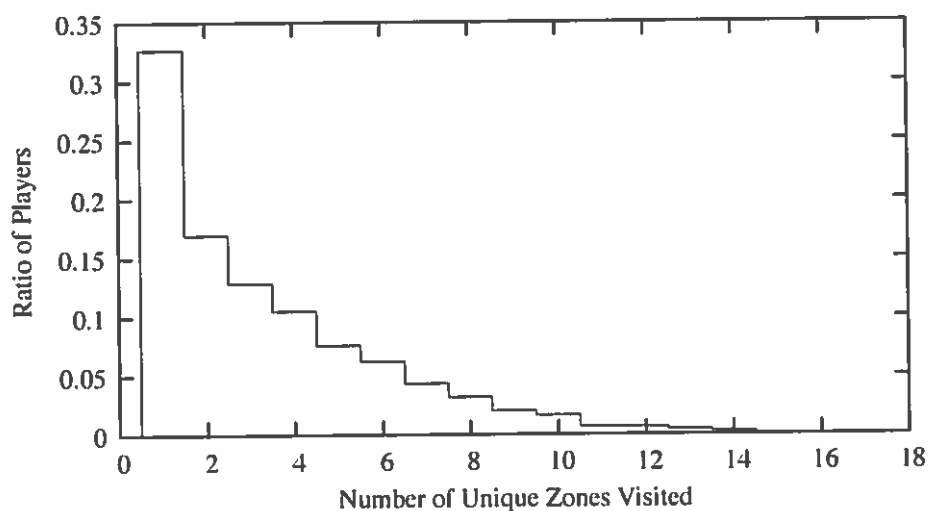


FIGURE 4.16: *Histogram of players versus number of unique zones visited:* This data shows how many unique zones a player visited during a playing session. As expected, a large portion of them (over 30%) only visited 1 zone, due to the high probability that a player only logs into the game for 15 minutes or less. Only about 15% visit more than 6 zones during a playing session.

When we measured the time spent in each zone, we saw that most players only spend around 15 minutes in each zone as shown in Figure 4.17. This is again due to the fact that most players only login to the game for 15 minutes or less. However, 35% spend 45 minutes or more in a zone. This indicates that players do not move from zone to zone very frequently.

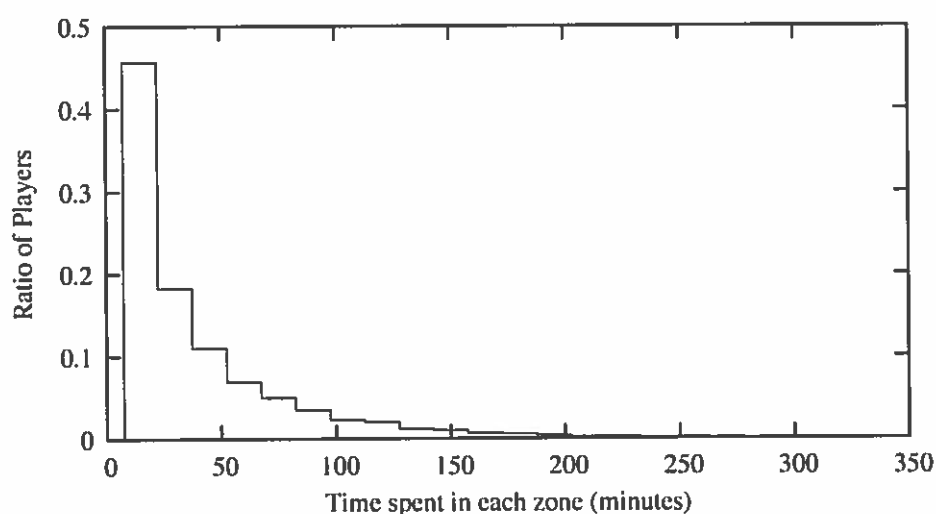


FIGURE 4.17: *Histogram of players versus time spent in a zone:* The histogram shows that, as expected, a large portion (approx. 45%) spend only 15 minutes in a zone, which is likely their entire time in their play session. On the other hand, 55% spend 30 minutes or more in a zone, and 35% spend 45 minutes or more in a zone before moving to another zone.

4.8.4 Measurement Summary

In summary, our measurement study provides us insight into the virtual populations and the behavior of players in an MMOG. The distribution of players over space appears to have a power-law relationship, with a few zones being highly populated, and most zones having a low population. The amount of time that players play the game, however, does not appear to follow a strictly power-law distribution. On the other hand, the amount of time spent per zone is also appears to have a power-law distribution, with most people spending one hour or less in a zone.

4.9 Simulation Experiments

In this section, we describe our experimental methodology and explain how we used the results from our measurement study to design our virtual world model and N-Tree protocol. We explain how our simulation works and describe the results, showing that N-Trees help scale the communication architecture.

The primary goal of our experiments is to demonstrate that the height in the N-Tree, when given an accurate model of the virtual world as seen in a modern MMOG, remains within a reasonable value so that event propagation experiences only a small delay. Knowledge of the tree height used in combination with forecasting, ensures N-Trees will cause minimum rollback and so the entire game will operate with the same playout latency experienced by the NEO groups at the leaf of the N-Tree.

Therefore, in our experiments we focus primarily on the average and worse-case heights of the N-Tree. In fact, the height of the N-Tree is the primary source of delay in event ordering and all other operations, such as the time to join the N-Tree and the amount of time we must forecast events is based on this factor. Since N-Trees do not attempt to balance themselves, theoretically performance of the N-Tree could be linear with respect to the number of nodes in the N-Tree. However, given even a power-law distribution of players, we show that the resulting height is compatible with realistic masking times associated with large scale events in MMOGs and user expectations.

4.9.1 Modeling a Virtual World

For our research, we decided to design a new virtual world simulation model. Previous models in prior research uniformly distributed players in the virtual world and then used a random way-point mobility model. The result of these two combinations is that players will tend to remain uniformly distributed in the virtual world. For N-Trees and other communication architectures, a uniform distribution of players (which we have shown to be unrealistic) provides an operating environment that is too idealized. We instead develop a new world model.

We make a few simplifying assumptions that we believe do not significantly affect the outcome of simulations with the model. First, we only model a 2-dimensional space and in fact our virtual world is a Cartesian square. Extrapolating data to a 3rd dimension is straightforward because popular locations will be simply distributed in a 3-dimensional space, instead of the 2-dimensional space. Second, players in the game are allowed 4 degrees of freedom, typical of a 2-dimensional world. We do not model movement restrictions.

To model player movement, we add some number of *hotspots* to the virtual world, which are popular locations. We model hotspots based on the results of our measurement study in Section 4.8, which showed that players were not uniformly distributed over the world, and instead were distributed according to a power-law model.

We use a Zipfian distribution to model player distributions because this type of power-law distribution is used to model population distributions on Earth. Zipf's law is stated mathematically as:

$$p_k(s, N) = \frac{1/k^s}{\sum_{n=1}^N 1/n^s} \quad (4.1)$$

In this equation, N is the number of elements, k their rank, and s is the exponent characterizing the distribution. For example, in our simulations we use 100 hotspots ($N = 100$) and we characterize the distribution with the exponent $s = .9$. We use 100 hotspots to have a similar scale to the number of zones in World of Warcraft. We then randomly chose 100 points in the virtual world and assign them a probability based on their rank.

To model player movement, each player has a location, a destination, and an amount of time they will wander in the vicinity of their destination before moving to a new destination. Destinations are chosen from the hotspots based on their assigned probability from the Zipf distribution. Initially, players choose their starting point from the hotspots also according to the Zipf distribution. They then travel to their location (in a straight line) each tick of the simulation clock. Once they arrive at their destination, they wander within a fixed radius of a hotspot for a given amount of time based on a uniform distribution (see Table 4.2). Finally, when this time expires, they choose a new hotspot to visit.

Figure 4.18 shows a snapshot from the visualization window of our simulator that is running a simulation with 100 hotspots and 3000 players on a 1000 by 1000 unit virtual world. Light circles are the hotspots and dark spots are the players. Some hotspots appear dark, but this is due to the large number of players currently visiting the area as derived from the Zipfian distribution.

4.9.2 Methodology

We developed a discrete simulator to study the performance of N-Trees by varying populations, numbers of hotspots, wandering times and virtual world sizes. The primary measurement we are interested in is the path length in the N-Tree. This is because, as our asymptotic analysis showed in Section 4.7, the height of the N-Tree dominates the cost of every action by the player, including joining the N-Tree, propagating events, and moving to new nodes in the N-Tree. Because N-Trees are not balanced, as a red-black tree might be, the maximum and average path lengths can affect performance adversely.

We ran the simulations for 10000 clock ticks, ignoring the first 1000 ticks to eliminate erroneous measurements from data dependent on starting conditions. Each simulation was repeated 10 times and the results were averaged. The variances in our averages within a 95% confidence interval allow us to conclude that our 10 runs were sufficient to be representative.

We summarize the simulation parameters for the experiments described in this section in Table 4.2. Note that we did perform experiments varying the number of hotspots, the size of the virtual world, and the wandering time. Increasing the number of hotspots, while keeping the virtual world size the same resulted in more uniformly distributed players, and visa-versa. Increasing the wandering time resulted in increasing the player density around hotspots, while decreasing the wandering time decreased player densities. For a virtual world of 1000 by 1000 units, we felt that 100 hotspots with a wandering time of 1000 units, sufficiently strained the system while realistically modeling game populations. We limited the maximum tree depth to 21 and we subdivided the leaves whenever they reached

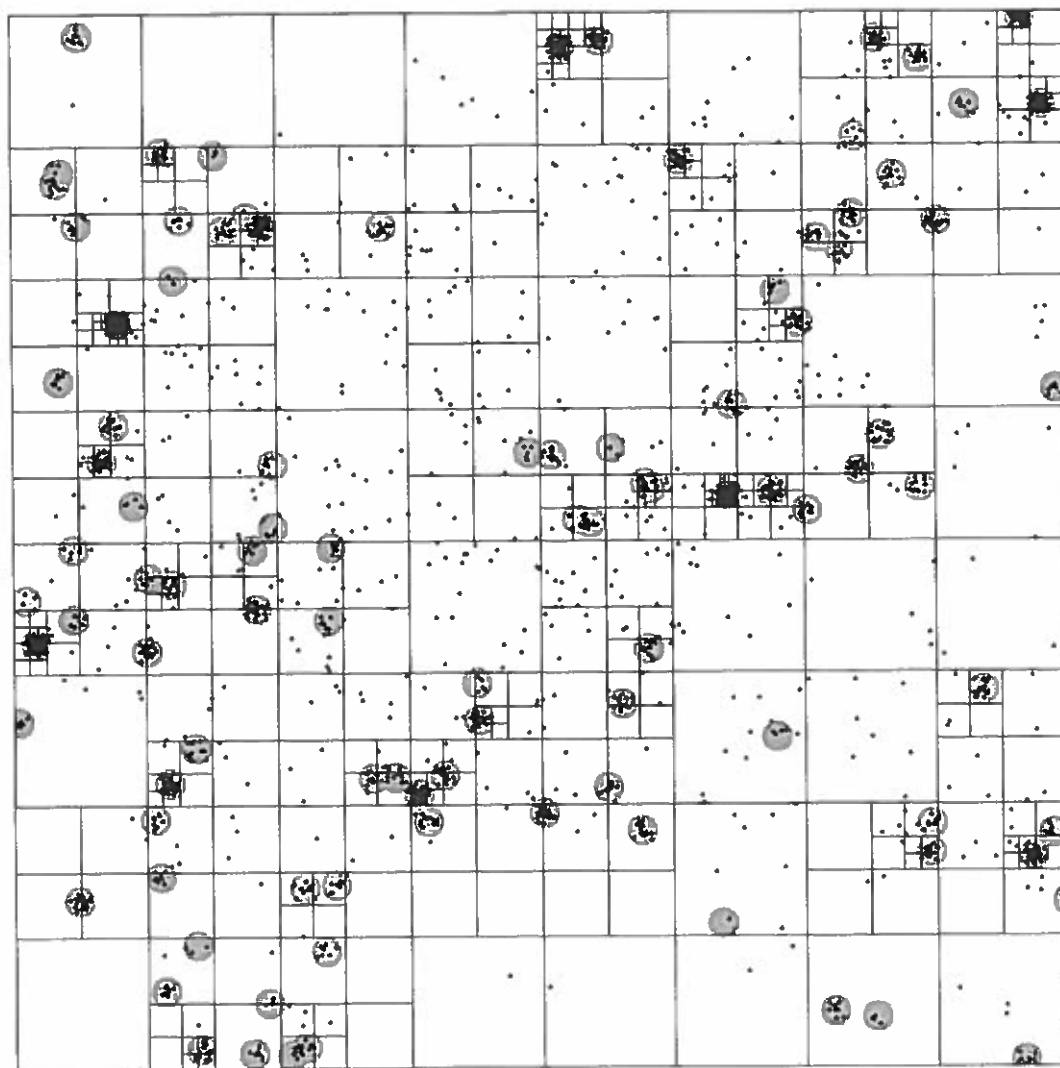


FIGURE 4.18: Visualization of *N-Tree* simulator: Large, lightly shaded circles represent hotspots, dark points represent players, and lines represent quadtree divisions. This example has 3000 players with 100 hotspots on a 1000 by 1000 unit virtual world.

TABLE 4.2: *Experiment parameters for N-Tree simulations.*

Variable	Value
Virtual world dimensions	1000 x 1000
Number of hotspots	100
Number of players	1,000 to 100,000
Wandering time	100
Total simulation time	10,000 units
Maximum leaf population	20
Maximum N-Tree depth	21
Zipfian values	$N = 100, k = .9$

more than 20 players. Note that if players were uniformly distributed and we completely filled the N-Tree to a depth of 21 at each leaf, our N-Tree would hold 87, 960, 930, 222, 080 players³.

4.9.3 Experimental Results

Our results show that the depth of the N-Tree resulting from a Zipfian distribution of hotspots in the application state space is on average $O(\lg n)$ with respect to the number of peers in the system. We measure the average and maximum path lengths given both a Zipfian and a uniform distribution of peers in the application state space. We use the uniform distribution as a base-line for performance because if the peers are uniformly distributed, the resulting tree will be optimally balanced.

Average Path Length in the N-Tree

In Figure 4.19 we see the results of our experiments when N-Trees are used with 100 to 100,000 players. This figure shows the average path length in the N-Tree. Given a Zipfian distribution of the populations of the hotspots, the path lengths in the N-Tree are approximately 2 hops longer than if players were uniformly distributed.

We now look at the histogram of path lengths for 1,000, 10,000, and 100,000 players. Figure 4.20 shows these three histograms. The histograms show the lengths of the paths

³ $(2^2)^{21} * 20$

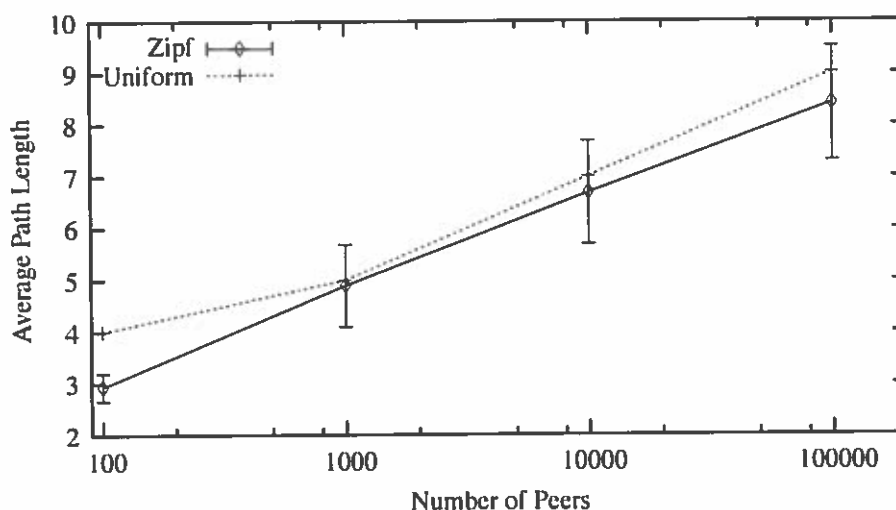
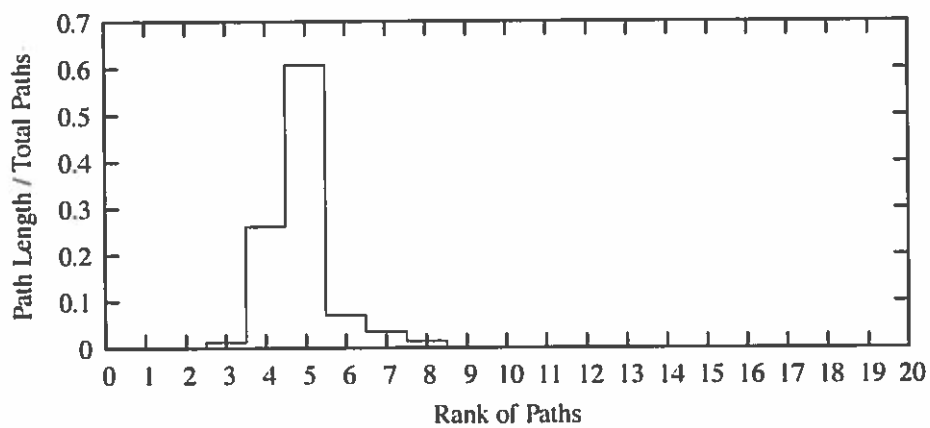


FIGURE 4.19: Average path lengths in the *N-Tree*: With 1,000, 10,000, and 100,000 players, the path lengths tend to be approximately 2 hops longer with the Zipfian distribution versus a uniform distribution.

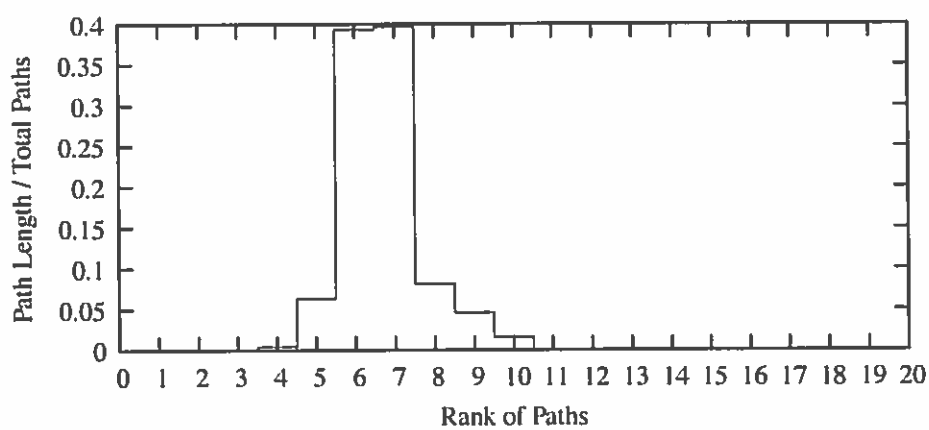
from the root of the *N-Tree* to the leaves. We are interested in the spread of the path lengths. For example, if we see that we have a bimodal distribution of path lengths, we know the *N-Tree* is very unbalanced. However, Figure 4.20 shows that most path lengths are clustered around an average, indicating that the *N-Tree* remains mostly balanced even with the Zipfian distribution of players.

The path lengths in the *N-Tree* indicate how far events could travel if their scopes exceed the leaf they were generated in. Empirical evidence from modern MMOGs demonstrates that most events in the system will be local to a small area and only a few large scale events will occur in the game. These small scale events typically include taking treasures, fighting monsters, or other similar actions.

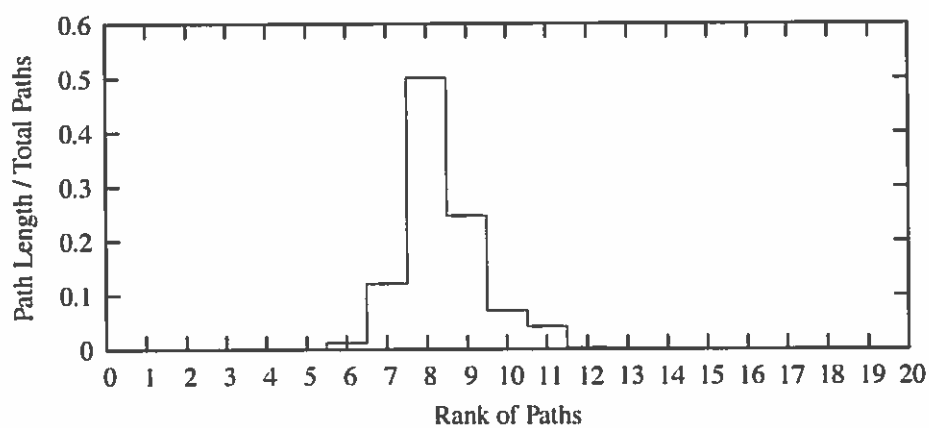
In Figure 4.21, we show how far events must be forecast if they are to reach all players in time. We make a simplifying assumption that nodes in the *N-Tree* have a 50ms one-way latency, though to date, no scientific measurements have been made that examine latencies to home users (the primary consumers of computer games). However, any two points in the United States can theoretically communicate within 20ms. Given the latency of electricity over copper wires and queuing delay, 50ms is a reasonable one-way latency.



(a) 1000 peers



(b) 10000 peers



(c) 100000 peers

FIGURE 4.20: Histogram of path lengths for 1,000, 10,000, and 100,000 peers.

The amount of time that it takes to forecast an update is equal to 6 times the height of the N-Tree. This is due to the need of the encrypted update to travel from a leaf to the root, and back to the leaf. Votes are then collected at the root and when the originating peer hears the final vote count, the plain-text update must then be sent to the other players. Thus, a $6 * 50ms = 300ms$ delay is added for each hop in the tree. Figure 4.21 shows that with up to 100,000 players, we will have to forecast events approximately 2.5 seconds on average. If the average delay between nodes in the N-Tree is 100ms, then we will have to forecast events approximately 5 seconds into the future.

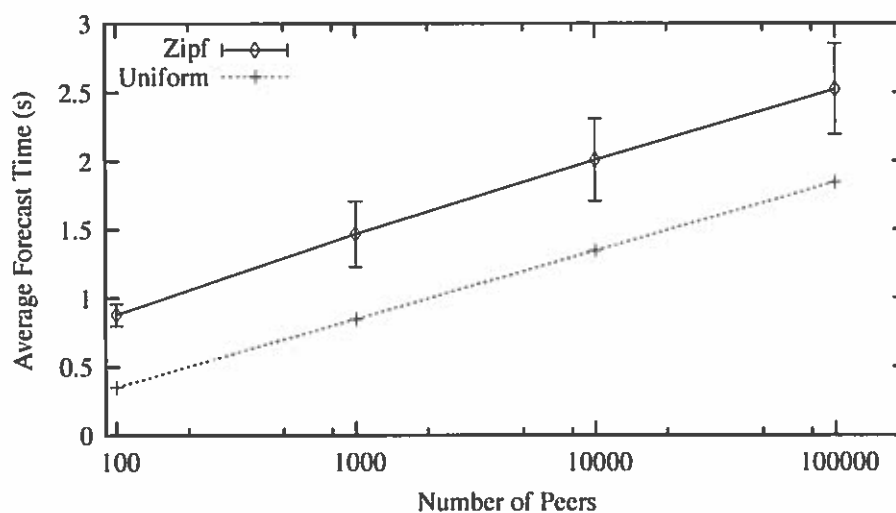


FIGURE 4.21: *Average forecasting time:* The amount of time a player must forecast an event on average to reach all peers in the game given 1,000, 10,000, and 100,000 players.

Maximum Path Length in the N-Tree

To examine the worse-case scenario, we look at the average maximum paths in the N-Tree. Figure 4.22 shows these values for 1,000, 10,000 and 100,000 players given both Zipfian and uniform distributions of players. In this worse-case scenario, the path length increases significantly more under the Zipfian distribution than under the uniform distribution. At 100,000 players, the path lengths for the Zipfian distribution are a maximum of approximately 20 hops, versus approximately 6 for the uniform distribution. This result shows that the distribution of players has an impact on the communication architecture;

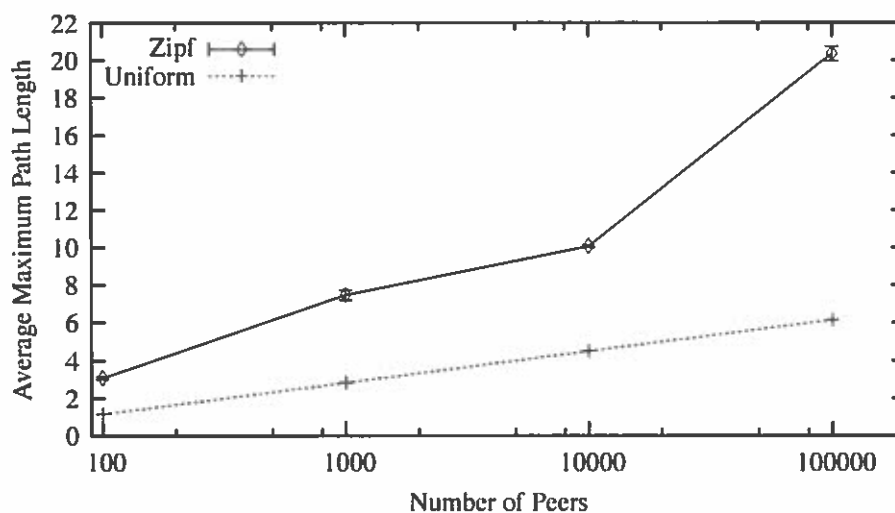


FIGURE 4.22: *Maximum path lengths in the N-Tree:* For 1,000, 10,000, and 100,000 players, the path lengths tend to increase significantly as we increase the number of players in the game. At 100,000 players, the maximum path length is over 14 hops longer than if we had a uniform distribution.

uniform distributions result in low maximum path lengths while Zipfian distributions have maximum path lengths 2 to 3 times that of the Zipfian distribution.

Extending this scenario to actual forecasting times, we plotted our results against a 50ms delay between nodes of the N-Tree as shown in Figure 4.23. Our results show that as the number of players increases in the game, the amount of time we have to forecast global-scale updates increases. At 100,000 players, we would need to forecast update over 20 seconds into the future. However, as we noted before, an update that affects 100,000 players cannot and should not be an instantaneous event in a peer-to-peer system. Increasing the delay to 100ms between nodes in the N-Tree would force players to forecast events 40 seconds into the future.

4.10 Conclusions

In this chapter, we created the N-Trees protocol, which is a multi-dimensional tree structure that recursively subdivides a space evenly along each dimension. N-Trees organize peers by their scope of interest in the virtual world. This organization allows peers to

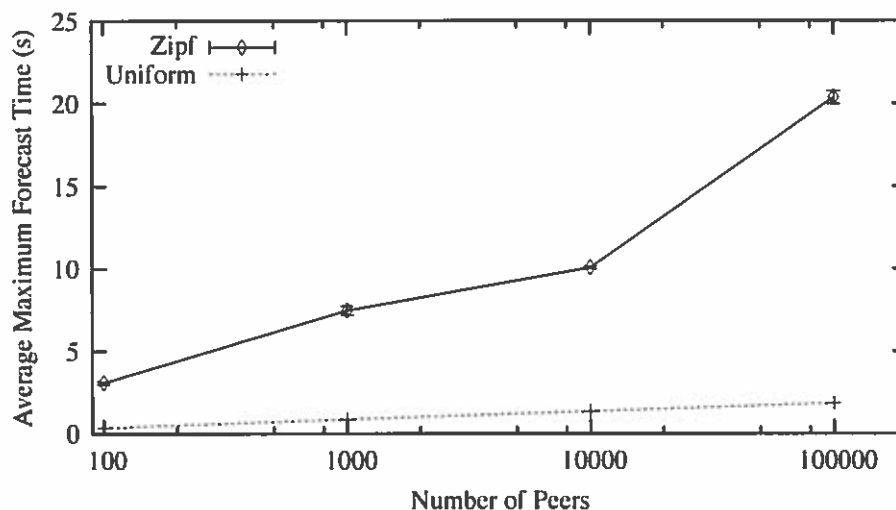


FIGURE 4.23: *Maximum forecasting time:* The amount of time a player must forecast an event in the worst-case to reach all peers in the game given 1,000, 10,000, and 100,000 players.

exchange events only with those peers close by. At the leaves, N-Trees use NEO, which provides majority game consistency for all local events. However, some updates have a scope which exceeds the leaf they were generated in, and therefore the updates must be ordered with other players within the scope of the update. Like NEO, N-Trees use majority voting to determine which updates are valid and which should be discarded.

We introduced the technique of forecasting events, a concept new to distributed systems. With this technique, we timestamp an update with a future time so that all interested peers have a chance to vote on accepting the update and so that it can arrive on time before it is scheduled to occur. With forecasting, rollback is minimized only to those players who did not receive the update on time due to packet loss or delay.

We created a new model of consistency, called majority★ game consistency. This model is similar to NEO's majority game consistency, except that the set of events accepted by the system are those which were voted on by a majority of players that were within the scope of each event. This majority can change from event to event and over time, but the system remains consistent.

As part of the analysis of N-Trees, we performed a measurement study to design an initial model for simulating virtual worlds and populations. In this study, we examined how populations fluctuate over time, the movement of players in time and space, and the distri-

bution of players over space. With further measurements and analysis, we plan to determine how these results fit to well known distributions such as the power-law distribution.

We then used the results from our measurement study to examine the height of N-Trees when simulated under our virtual world model. The results show that the height of the N-Tree is logarithmic, given the power-law distributions of players that we used. This result is important because it shows that the time needed to forecast updates is within the bounds of latencies already present in large scale events for MMOGs and player expectations.

In summary, our work with N-Trees has three important contributions. First, we developed the N-Tree protocol which helps scale NEO to hundreds of thousands of players. Second, we developed a new consistency model for games that uses majorities and scopes to determine which events are valid. Third, we performed the first measurement study of virtual populations in MMOGs which will help develop future models for the simulation of MMOGs.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

In this dissertation, we investigated the problem of scalable, cheat-proof and real-time event ordering for RIM applications. We focused our work on MMOGs, an important example of RIM applications, and developed the NEO and N-Tree protocols. NEO provides cheat-proof and real-time event ordering through a majority voting system and cryptography. We proved that NEO provides majority game consistency and prevents peers from cheating. We further demonstrated NEO's playability as a game protocol through simulations, showing that NEO is capable of delivering updates with a sufficient update frequency, low playout latency, and little rollback under a range of network conditions and for all game archetypes. However, NEO is limited in scalability and thus we developed N-Trees, which use NEO at the leaves, to address this problem.

The N-Tree protocol organizes uses a combination of hierarchical organization, majority voting, event scoping, and forecasting to provide scalable, peer-to-peer communications for games. N-Trees organize peers hierarchically by their scope of interest in the virtual world. Updates are propagated over the tree according to their scopes, and peers forecast updates based on the height of the N-Tree, thereby minimizing rollback. We showed that N-Trees provide majority★ game consistency for all players in the game. We analyzed N-Trees to show that their theoretical performance was acceptable for MMOGs.

Because the true performance of N-Trees is dominated by height of the tree resulting from the distribution of players in the virtual world, we designed a set of simulations that measured the average and maximum heights of the tree. We first performed an initial measurement study and concluded that player distributions appeared to follow a power-law

distribution. We then simulated N-Trees with this knowledge to measure their average and maximum heights over time. The height of the N-Tree remained logarithmic in the number of nodes in the N-Tree at populations of up to 100000 players.

Both NEO and N-Trees advance the state-of-the-art in protocols for peer-to-peer games. Together, they provide a holistic solution to the problem of scalability, cheating, and real-time event ordering while prior work only addressed one or two of these three issues.

In our investigations, we also examined previous consistency models for distributed systems and adapted these for RIM applications. Lamport's sequential consistency [8] is too restrictive for RIM applications due to the requirement that all events from all participants would need to be seen in a single sequential order, introducing significant network delay to achieve this goal.

As an alternative to sequential consistency, we designed two models of consistency specific to multiplayer games: majority game consistency and majority★ game consistency. With majority game consistency, only those updates accepted by a majority of players occur in the sequence of events seen by players. Majority★ game consistency follows the same logic of majority game consistency, except that only the players within the scope of an event can accept an update. The advantage with majority★ game consistency is that only a subset of all players need to accept an update, though that subset are those players which fall within the scope of the event.

Applying our consistency models towards more general distributed systems results in two consistency models: *majority consistency* and *majority★ consistency*, which are not specific to games. Majority consistency was defined in Chapter III. Majority★ consistency is defined as:

Definition 5.0.1. *Majority★ consistency*: the result of any computation is the same as if all reads and writes follow a sequential order, and only the reads and writes seen by a majority of the systems within the scope of those reads and writes appear in that sequence and in the order specified by its program.

Under this consistency model, a read or write is only accepted if a majority of the systems accepted the read or write, where the majority is taken from those systems which fall under the scope of the read and write. This allows the distributed system to ignore nodes

which may have failed or may be experiencing poor network conditions so that computations may continue. The primary drawback is that those systems in the minority will need to rollback state when they discover what set of reads and writes the majority has agreed on.

To understand the direct impact of our research, we present a simple RIM application that would benefit from it. Most students will never have the chance to visit the space station and a simulation of it would give students the ability to run experiments, interact with each other, and operate the space station. However, most schools do not have the budget to design, deploy, and host a large-scale simulation for their students for educational purposes. Using a peer-to-peer architecture, schools could pool their resources to provide immersive educational environments for students across the world. Our communication architecture would provide scalability, reliability, reduced bandwidth, and probably most importantly, more affordability at each site.

Beyond the direct impact of our research on MMOGs, our research makes other RIM applications more viable because it addresses problems of scalability, real-time messaging and cheating. Thus, distance learning applications, collaborative applications, and immersive educational and training simulations can be designed using our protocols with guarantees on performance and security and capable of working over best-effort networks such as the Internet. In addition, the long range impact of our work is on the fundamental work in distributed systems. We designed new models of consistency which can be applied in new ways to distributed systems.

As for future work, we divide it into two categories: work related to NEO and N-Trees and related future studies. The NEO and N-Tree future work includes:

- *Improving the overhead of NEO:* The main factor that reduces NEO's scalability is its overhead for cheat-prevention and consistency. If we can reduce this, we could increase NEO's scalability.
- *Implementing NEO in a modern game and studying its performance:* We have not experimented with NEO in a modern computer game. Examining its behavior in this situation would give us a complete understanding of its performance.

- *Investigating the problem of collusion:* With NEO, we assume that a majority of players cannot collude to gain an artificial majority. We would like to investigate techniques to reduce or eliminate collusion among players.
- *Eliminating the need for rounds with NEO:* In NEO, we use rounds to determine when hashes and updates should be sent out. However, we may be able to eliminate rounds by allowing voting to occur once each update is received. The security implications of such a technique would need to be studied further.
- *Continued measurement and analysis of data from virtual populations in MMOGs:* Our current measurement study needs to gather more data to get a more complete picture of the behavior of virtual populations. We need to fully understand how players are distributed in the virtual world and to develop a model that takes arrival rates, player movement, session times, and player movement into account.
- *Further analysis of N-Trees with events of varying scopes:* Our current analysis looks primarily at the height of N-Trees because it can significantly influence the delay for almost all operations in the tree. On the other hand, if we simulate various scopes, we can more accurately determine how well forecasting works, how much rollback occurs, and how packet loss and delay affects event ordering over N-Trees.
- *N-Tree mapping to DHTs:* We have examined how N-Trees can easily map to CAN, however we should be able to map it to any DHT [11]. Thus, we would like to understand an effective and efficient method for mapping nodes in the N-Tree to various DHTs.
- *Studying the effects of churn on NEO and N-Trees:* Electing leaders and adding or removing players from a NEO group or an N-Tree can affect game play. Studying these effects and deciding whether they are significant is important to the analysis of NEO and N-Trees.
- *Understanding the distribution of scope sizes of events:* To date, no studies have looked at the distribution of scope sizes in events. Understanding these distributions would allow us to more accurately model games.

In addition, the we see the following three problems as future related studies:

- *Different peer-to-peer structures for event ordering:* N-Trees suffer from the fact that trees are typically a weak structure in terms of resilience. A single broken branch can disconnect a subtree. While we address this problem through the use of multiple N-Trees, other structures may provide more resilient message delivery while remaining efficient.
- *Mapping the shared state of other applications to N-Trees:* While not all applications are capable of hierarchically decomposing their shared state, we would like to investigate the use of N-Trees with other applications as a foundation to make them more scalable.
- *Applying majority consistency to peer-to-peer computing:* Distributed systems on the scale of SETI@home [84] have tens of thousands of nodes in the system. However, these systems rely on the fact that their tasks are easily decomposed into small subtasks that do not require inter-node communication. Our notion of majority consistency could be applied in peer-to-peer computing to reduce communication and use it for more general purpose computing.
- *Peer-to-peer scheduling:* RIM applications should be able to take advantage of all the peers in the system by scheduling processes to be executed on them. Using distributed MMOGs as an example, we could schedule tasks such as artificial intelligence on multiple systems, allowing characters in the game to interact with players in a more realistic manner than current client/server architectures provide. Like our communication architecture, we are faced with problems of reliability, timeliness, and security.

In summary, this dissertation has several important contributions. First, we advance the state-of-the-art in peer-to-peer protocols for large-scale, distributed games with NEO and N-Trees. These are the first protocols which address scalability, cheating, and consistency with peer-to-peer networking for games. Future research in the design of distributed MMOGs will be able to leverage our work and solve other related problems. In addition,

our protocols may be used for other RIM applications. Second, we performed the first measurement study of virtual population characteristics and behavior in MMOGs. Prior work has examined network characteristics, which though important, do not allow us to accurately simulate virtual world to analyze future protocols. Last, we developed new consistency models that apply to games and to general distributed systems. Thus, majority consistency and majority★ consistency can be used in the development of peer-to-peer computing architectures to accommodate packet loss and latency typical on the Internet.

BIBLIOGRAPHY

- [1] "EverQuest." [Online]. Available: <http://www.everquest.com>
- [2] D. Kushner, "Engineering EverQuest," *IEEE Spectrum*, July 2005.
- [3] N. E. Baughman and B. N. Levine, "Cheat-proof payout for centralized and distributed online games," in *Proceedings of IEEE Infocom*, 2001, pp. 104–113.
- [4] A. R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: Supporting scalable multi-attribute range queries," in *Proceedings of ACM SIGCOMM*, August 2004.
- [5] C. GauthierDickey, V. Lo, and D. Zappala, "Using N-Trees for scalable event ordering in peer-to-peer games," in *Proceedings of NOSSDAV*, June 2005.
- [6] B. Knutsson, H. Lu, W. Xu, and B. Hopkins, "Peer-to-peer support for massively multiplayer games," in *Proceedings of IEEE Infocom*, March 2004.
- [7] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.
- [8] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Computers*, pp. 690–691, September 1979.
- [9] P. Bettner and M. Terrano, "1500 archers on a 28.8: Network programming in the Age of Empires and beyond," in *Game Developers Conference*, March 2001.
- [10] "World of Warcraft." [Online]. Available: <http://www.worldofwarcraft.com>
- [11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *Proceedings of ACM SIGCOMM*, 2001, pp. 161–172.
- [12] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of ACM SIGCOMM*, 2001, pp. 149–160.
- [13] Y. Chu, S. Rao, S. Seshan, and H. Zhang, "Enabling conferencing applications on the internet using an overlay multicast architecture," in *Proceedings of ACM SIGCOMM*, 2001, pp. 55–67.

- [14] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron, "SCRIBE: A large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in Communications (JSAC)*, 2002.
- [15] C. Diot and L. Gautier, "A distributed architecture for multiplayer interactive applications on the internet," *IEEE Networks magazine*, vol. 13, no. 4, July/August 1999.
- [16] M. Claypool and K. Claypool, "Latency and player actions in online games," *Communications of the ACM*, vol. 49, no. 11, pp. 40–45, 2006.
- [17] L. Lamport, "The part-time parliament," *ACM Trans. on Comp. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.
- [18] K. Birman, "Replication and fault-tolerance in the isis system," in *10th ACM Symposium on Operating System Principles*, December 1985, pp. 79–86.
- [19] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin, "RTCAST: Lightweight multicast for real-time process groups," in *Proceedings of IEEE Real-Time Technology and Applications Symposium*, 1996.
- [20] S. E. Deering and D. R. Cheriton, "Multicast routing in datagram internetworks and extended LANs," *ACM Transactions on Computer Systems*, vol. 8, no. 2, pp. 85–110, 1990.
- [21] Y. Chu, S. G. Rao, and H. Zhang, "A case for end system multicast," in *Proceedings of ACM SIGMETRICS*, 2000, pp. 1–12.
- [22] S. Ratnasamy, M. Handley, R. Karp, and S. Shenkar, "Application-level multicast using content addressable networks," in *Proceedings of Network Group Communications*, November 2001.
- [23] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz, "Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination," in *Proceedings of ACM NOSSDAV*, June 2001.
- [24] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast," in *Proceedings of ACM SIGCOMM*, 2002, pp. 205–217.
- [25] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design, 3rd Edition*. Addison-Wesley, 2001.
- [26] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.
- [27] "The network time protocol." [Online]. Available: <http://www.ntp.org>
- [28] M. Dubois, C. Scheurich, and F. A. Briggs, "Synchronization, coherence, and event ordering in multiprocessors," *IEEE Computer*, vol. 21, no. 2, pp. 9–21, 1988.

- [29] I. Keidar and S. Rajsbaum, "On the cost of fault-tolerant consensus when there are no faults—a tutorial. MIT-LCS-TR-821," MIT, Massachusetts Institute of Technology, Cambridge, MA, 02139, Tech. Rep. MIT-LCS-TR-821, May 2001.
- [30] A. Fekete, M. Kaashoek, and N. Lynch, "Implementing sequentially consistent shared objects using broadcast and point-to-point communication," *Journal of the ACM*, 1998.
- [31] "IEEE Standard for Distributed Interactive Simulation – Application Protocols (IEEE STD 1278.1-1995)," IEEE Computer Society, 1995.
- [32] "IEEE Standard for Distributed Interactive Simulation – Communication Services and Profiles (IEEE Std 1278.2-1995)," IEEE Computer Society, 1995.
- [33] A. Pope, "The SIMNET network and protocols," BBN Systems and Technologies, Cambridge, MA, Tech. Rep. 7102, July 1989.
- [34] K. Almeroth, "The evolution of multicast: From the Mbone to inter-domain multicast to Internet2 deployment," *IEEE Network*, vol. 14, pp. 10–20, Jan/Feb 2000.
- [35] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei, "An architecture for wide-area multicast routing," in *Proceedings of ACM SIGCOMM*, 1994, pp. 126–135.
- [36] T. Ballardie, P. Francis, and J. Crowcroft, "Core based trees (CBT)," in *Proceedings of ACM SIGCOMM*, 1993, pp. 85–95.
- [37] K. L. Calvert, E. W. Zegura, and M. J. Donahoo, "Core selection methods for multicast routing," in *Proceedings of IEEE ICCCN*. Las Vegas, Nevada: IEEE, September 1995, pp. 638–642. [Online]. Available: citeseer.ist.psu.edu/article/calvert95core.html
- [38] D. Thaler and C. V. Ravishankar, "Distributed center-location algorithms," *IEEE Journal on Selected Areas in Communications*, vol. 15, no. 3, pp. 291–303, 1997. [Online]. Available: citeseer.ist.psu.edu/thaler97distributed.html
- [39] S. Kumar, P. Radoslavov, D. Thaler, C. Alaettinoğlu, D. Estrin, and M. Handley, "The MASC/BGMP architecture for inter-domain multicast routing," in *Proceedings of ACM SIGCOMM*, 1998, pp. 93–104.
- [40] H. W. Holbrook and D. R. Cheriton, "IP multicast channels: EXPRESS support for large-scale single-source applications," in *Proceedings of ACM SIGCOMM*, 1999, pp. 65–78.
- [41] D. Zappala and A. Fabbri, "Using SSM proxies to provide efficient multiple-source multicast delivery," in *Proceedings of IEEE Globecom*, November 2001.

- [42] H. W. Holbrook, S. K. Singhal, and D. R. Cheriton, "Log-based receiver-reliable multicast for distributed interactive simulation," in *Proceedings of ACM SIGCOMM*, 1995, pp. 328–341.
- [43] J. C. Lin and S. Paul, "RMTP: A reliable multicast transport protocol," in *Proceedings of IEEE Infocom*, March 1996, pp. 1414–1424.
- [44] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang, "A reliable multicast framework for light-weight sessions and application level framing," *IEEE/ACM Transactions Networking*, vol. 5, no. 6, pp. 784–803, 1997.
- [45] C.-G. Liu, D. Estrin, S. Shenker, and L. Zhang, "Local error recovery in SRM: Comparison of two approaches," *IEEE/ACM Transactions on Networking*, vol. 6, no. 6, pp. 686–699, 1998.
- [46] Y. Chawathe, S. McCanne, and E. A. Brewer, "RMX: Reliable multicast for heterogeneous networks," in *Proceedings of IEEE Infocom*, 2000.
- [47] S. K. Kasera, G. Hjálmtýsson, D. F. Towsley, and J. F. Kurose, "Scalable reliable multicast using multiple multicast channels," *IEEE/ACM Transactions on Networking Netw.*, vol. 8, no. 3, pp. 294–310, 2000.
- [48] V. Jacobson, "Congestion avoidance and control," in *Proceedings of ACM SIGCOMM*, 1988, pp. 314–329.
- [49] S. Bhattacharyya, D. Towsley, and J. Kurose, "The loss path multiplicity problem for multicast congestion control," in *Proceedings of IEEE Infocom*, 1998, pp. 856–863.
- [50] S. J. Golestani and K. K. Sabnani, "Fundamental observations on multicast congestion control in the internet," in *Proceedings of IEEE Infocom*, 1999, pp. 990–1000. [Online]. Available: citeseer.ist.psu.edu/golestani99fundamental.html
- [51] S. McCanne, V. Jacobson, and M. Vetterli, "Receiver-driven layered multicast," in *Proceedings of ACM SIGCOMM*, 1996, pp. 117–130.
- [52] I. Rhee, N. Ballaguru, and G. Rouskas, "MTCP: Scalable tcp-like congestion control for reliable multicast," in *Proceedings of IEEE Infocom*, March 1999.
- [53] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277–288, Nov 1984.
- [54] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*. Springer-Verlag, 2001, pp. 329–350.

- [55] "Napster," <http://www.napster.com>.
- [56] "Gnutella," <http://www.gnutella.com>.
- [57] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, January 2004.
- [58] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao, "OceanStore: An architecture for global-scale persistent storage," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 190–201.
- [59] L. Gautier, C. Diot, and J. Kurose, "End-to-end transmission control mechanisms for multiparty interactive applications on the internet," in *Proceedings of IEEE Infocom*, 1999.
- [60] E. Cronin, B. Filstrup, and S. Jamin, "Cheat-proofing dead reckoned multiplayer games," in *International Conference on Application and Development of Computer Games*, January 2003.
- [61] C. Chambers, W. Feng, S. Sahu, and D. Saha, "Measurement-based characterization of a collection of online games," in *Proceedings of the Internet Measurement Conference*, 2005.
- [62] A. R. Bharambe, S. Rao, and S. Seshan, "Mercury: A scalable publish-subscribe system for internet games," in *Proceedings of ACM NetGames*, April 2002.
- [63] S. Rooney, D. Bauer, and R. Deydier, "A federated peer-to-peer network game architecture," *IEEE Communications*, vol. 42, no. 5, May 2004.
- [64] "NIST Net." [Online]. Available: <http://snad.ncsl.nist.gov/nistnet>
- [65] J. Nichols and M. Claypool, "The effects of latency on online madden nfl football," in *Proceedings of ACM NOSSDAV*, June 2004.
- [66] K. Li, S. Ding, and D. McCreary, "Analysis of state exposure control to prevent cheating in online games," in *Proceedings of ACM NOSSDAV*, June 2004.
- [67] Ensemble Studios, "Age of Empires," <http://www.ensemblestudios.com/aoe.htm>.
- [68] P. W. Hutto and M. Ahamad, "Slow memory: Weakening consistency to enhance concurrency in distributed shared memories," in *Proceedings of the 10th International Conference on Distributed Computing Systems*, May 1990, pp. 302–311.

- [69] D. Mosberger, "Memory consistency models," University of Arizona," TR 93/11, 1993.
- [70] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.
- [71] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 3, pp. 404–425, 1985.
- [72] "The ns-2 simulator." [Online]. Available: <http://www.isi.edu/nsnam/ns/>
- [73] W. chang Feng, F. Chang, W. chi Feng, and J. Walpole, "Provisioning on-line games: A traffic analysis of a busy counter-strike server," in *Internet Measurement Workshop*, 2002.
- [74] J. Kim, J. Choi, D. Chang, T. Kwon, Y. Choi, and E. Yuk, "Traffic Characteristics of a Massively Multiplayer Online Role Playing Game," in *Proceedings of ACM NetGames*, 2005.
- [75] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics: Principles and Practice*, 2nd ed. Addison-Wesley, 1997.
- [76] B. Zhang, S. Jamin, and L. Zhang, "Host multicast: A framework for delivering multicast to end users," in *Proceedings of IEEE Infocom*, June 2002.
- [77] A. St. John and B. N. Levine, "Supporting P2P games when players have heterogeneous resources," in *Proceedings of ACM NOSSDAV*, June 2005.
- [78] D. J. V. Hook, J. O. Calvin, M. K. Newton, and D. A. Fusco, "An Approach to DIS Scalability," in *11th DIS Workshop*, September 1994.
- [79] E. Léty and T. Turletti, "Issues in Designing a Communication Architecture for Large-Scale Virtual Environments," in *Proceedings of Network Group Communications*, 1999, pp. 54–71.
- [80] "XPilot," <http://www.xpilot.org>.
- [81] S. A. Tan, W. Lau, and A. Loh, "Networked Game Mobility Model for First-Person Shooter Games," in *Proceedings of ACM NetGames*, 2005.
- [82] "The LUA Programming Language." [Online]. Available: <http://www.lua.org>
- [83] "Realm status page for World of Warcraft." [Online]. Available: <http://www.worldofwarcraft.com/realmstatus/>
- [84] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: An experiment in public-resource computing," *Communications of the ACM*, vol. 45, pp. 56–61, 2002.