A Distributed Multiplayer Game Server System

Eric Cronin Burton Filstrup Anthony Kurc
Electrical Engineering and Computer Science Department
University of Michigan
Ann Arbor, MI 48109-2122
{*ecronin,bfilstru,tkurc*}@*eecs.umich.edu*

May 4, 2001

Chapter 1

Introduction

1.1 Background

Online multiplayer games have improved dramatically in the past few years. The newest online virtual worlds such as Everquest [5] feature fantastic artwork, realistic graphics, imaginative gameplay and sophisticated artificial intelligence. Before the next generation of multiplayer games can complete the transition to lifelike virtual worlds, they must be able to support real-time interactions. The main obstacle to real-time interactions is the Internet's inability to provide low-latency guarantees. In this paper, we present a system for enabling real-time multiplayer games.

In particular, we focus on real-time multiplayer games that have strong consistency requirements. That is, all players must share a common view of a complex virtual world. The combination of low latency and absolute consistency is difficult to achieve because messages may be delayed indefinitely in the network. Our system realizes these goals using three building blocks: a Mirrored-Server architecture, a trailing state synchronization protocol, and a low-latency reliable multicast protocol. We have implemented and integrated these three components into a working prototype and performed some preliminary experiments on our system.

Commercial games are either designed on top of client-server architectures or, less frequently, on top of peer-to-peer architectures. Client-server architectures are simple to implement and allow gaming companies to retain control over the game state. Peer-to-peer architectures deliver lower latencies and eliminate bottlenecks at the server. Our proposed Mirrored-Server architecture requires a complex consistency protocol, but it achieves low latency and allows administrative control over the game state.

In order to make our mirrored server architecture attractive, we have developed a consistency protocol that can be used to port a client-server game to a Mirrored-Server architecture with minimal modifications. The two key pieces of the consistency protocol are a synchronization mechanism called trailing state synchronization and CRIMP, a reliable multicast protocol.

Previous synchronization mechanisms such as bucket synchronization and Time Warp are not well suited to the demands of a real-time multiplayer game. Trailing state synchronization is designed specifically for real-time multiplayer games, and achieves better responsiveness and scalability while maintaining near-perfect consistency.

The consistency protocol assumes a low-latency, many-to-many reliable multicast protocol. Previous reliable multicast protocols focus more on scalability

of state rather than overall packet latency. CRIMP is a custom-built, application-specific multicast protocol that provides the infrastructure requirements of our architecture in a low latency manner.

Two bootstrapping services will be included in the final system, although they have not yet been implemented. The first is a master server that allows users to locate candidate game servers. The second is a server selection service named qm-find that can be used to locate the closest game server to a particular client.

The most significant result of this project is a fully operational, distributed Quake system, complete with a working synchronization mechanism and a functional multicast protocol. In addition, we have established the infrastructure to test and evaluate our system under different network conditions. The experimental evaluation of trailing state synchronization is in its early phases, and our preliminary results do not adequately explore the mechanism's potential to respond to different network conditions. In future work, we would like to concentrate on understanding the effects of different configurations on trailing state synchronization's performance.

1.2 Quake

We chose Quake as our proof-of-concept game because it is very sensitive to latency and provides a virtual world environment. Quake is a commercial game that was not designed for a distributed architecture. As such, Quake forces us to address the needs of a real game, as opposed to a game that fits neatly into our architecture. Another reason for using Quake is that it is one of the few commercial games for which the source code has been released.

In this section, we introduce Quake and describe the features of Quake that are relevant to our report. We outline Quake's architecture, including its master server. We describe Quake's interest management techniques and then delve into details about the messaging protocols.

1.2.1 Introduction

Quake is a 3-D first player shooter, by id Software, the creators of a revolutionary and spectacularly successful line of games including Wolfenstein 3D, Doom, Quake and Quake III. Besides 3-D graphics, their major innovation was the online multiplayer game. College students found that they could put the newly installed dorm LAN's to use by engaging in mortal combat with their neighbors. Multiplayer Quake is a simple game. Your avatar has a gun and so does everyone else. You're going to die in the next minute or so, but before that happens, your goal is to finish off as many of your opponents as possible. All interactions take place in real-time, and more than a 100 ms lay can be a crippling handicap.

John Carmack, the mastermind behind Quake, released the source code in 1999. Two open source projects arose to improve Quake, QuakeForge and QuakeWorld. The two subsequently merged their sources into Quakeforge, which has developed several new versions of the original game. Development in Quakeforge is sporadic, with a handful of people doing most of the work. Their major contributions have been porting Quake to new platforms, supporting more sound and graphics hardware, and enhancing the graphics. Quakeforge has also built in some simple cheat detection algorithm and a denial-of-service attack prevention mechanism.

A significant amount of information about Quake has been compiled by the Quake Documentation Project. Two pieces of this project describe the network code: The Unofficial Quake Specification and The Unofficial DEM Format Description. For the

most part, these documents focus on the details of the message formats. Neither is a comprehensive look at the networking issues in Quake. Other Quake resources can be found at the Quake Developer's Pages.

The purpose of this document is to give a high-level overview of the technologies Quake uses to enable multiplayer games. The first section describes the Quake Client-Server architecture. Next, three optimizations are discussed: dead reckoning, area of interest filtering and packet compression. The Quakeforge modifications to prevent cheating and denial-of-service attacks are discussed. Finally, the last section presents the network protocols.

1.2.2 Architecture

Quake uses a centralized Client-Server architecture. In this architecture, the server is responsible for computing all game state and distributing updates about that game state to the client. The client. The client acts as a viewport into the server's state and passes primitive commands such as button presses and movement commands to the server. Because this architecture allows most of the game state computation to be performed on the server, there is no consistency problem. However, the gameplay is very sensitive to latencies between the client and the server, as users must wait at least a round trip time for key presses to be reflected in reality.

The server is responsible for receiving input commands from client, updating the game state, and sending out updates to the clients. The control loop for this procedure is discussed in more detail in Appendix A. The server's biggest job is keeping track of entities. An entity is almost any dynamic object: avatars, monsters, rockets and backpacks. The server must decide when a player is hit, use physics models to calculate the position of free-falling objects, turn key presses into jumps and weapon fires, etc. The server also informs clients of world state: gravity, lighting and map information. Aside from its role in gameplay, the server is a session manager that starts and ends games, accepts client connections and performs numerous other administrative tasks.

The client sends simple avatar control commands (movements, tilts, buttons) to the server. It uses its cache of the game state plus any updates from the server to render a 3-D representation of the virtual world. The graphics rendering is performed on the client, as is fine-grained collision detection. The client has been the main area of interest in the game developer community because it is where all the 'cool graphics' are done. However, from an architectural or networking point of view, the Quake client is relatively simple compared to the server.

1.2.3 Master Server

Quake supports a master server that keeps track of active game servers on the Internet. Although the master server is not packaged with Quake, GameSpy provides a full-featured master server service. For a Quake server to be listed in GameSpy's database, the administrator must manually add GameSpy's IP address to the master server list. The server then periodically sends heartbeats containing the number of connected client to the master server. The server also informs the master server when it is shutting down.

1.2.4 Area of Interest Filtering

One of the advantages of having a server architecture is that the server can dynamically filter information before sending it to the clients. The Quake server only sends a client information for entities that are within that player's potential field of sight. Similarly, sounds are filtered to those that are within that player's earshot.

Calculating an entity's potential field of sight is

tricky. The data structure used to do this is called a Binary Space Partition (BSP) tree or octree. The three dimensional world is split into eight regions by three planes. Each region can be further subdivided until the necessary resolution is achieved. The field of sight for entities in each leaf of the BSP tree is precomputed so that the server can quickly determine which other leaves of the BSP tree to include in the filter. For example, a room might be represented by a single leaf. This leaf would have associated with it a list of adjacent rooms (leaves) within the room's field of sight. A great deal of information about this general technique is available on the web. Information about BSP trees in Quake is available in the Unofficial Quake Specs.

1.2.5 Network Protocols

Almost all of the messages exchanged between the client and the server during gameplay are sent unreliably. Because there are few reliable messages, the client and server can get away with an inefficient yet simple stop-and-wait protocol with single-bit sequence numbers. Although recovery of lost packets is not necessary, the server does require knowledge of which packets have been dropped so that it can implement the FEC scheme described above. This is accomplished using sequence numbers.

Both the client and the server have a frame rate. On the client, this frame rate is the frame rate of the rendered scene. On the server, a frame is a single iteration through the receive, process, send loop. This is typically a tight loop unless the server machine is also running a client, in which case some time is reserved for client-side processing. At the start of a frame, all waiting packets are read. During the processing step of a frame, a number of different functions may write messages into either the reliable or unreliable message buffers. At the end of each frame, the server sends at most one packet to each client. Similarly, the client sends at most one packet to the server. Packets can contain both reliable and unreliable data. Reliable data takes precedence over unreliable data when constructing a packet. All unsent data is sent in the next frame.

Reliability is achieved using a stop-and-wait protocol with a single bit reliable sequence number. The receiver will not reorder packets; stale packets are discarded. Acknowledgments are piggybacked on outgoing messages. Both unreliable and reliable messages are sent using UDP datagrams.

Congestion control for both unreliable and reliable messages is a simple rate-based scheme. In Quakeforge 5, the channel is assumed to have a capacity of 20 Kbps. However, the original scheme used round trip times as a measure of bandwidth, allowing the channel capacity to increase up to 40 Kbps. The chosen rate is enforced by aborting packet transmissions that might result in an overloaded channel. A packet is only sent if the sender decides, based on the time and size of the last transmission, that the channel is currently clear. Aborted packets are sent at the end of the next frame. No flow control is performed - the sender assumes that the receiver can process packets as fast as the sender can send them.

1.2.6 Client to Server Messages

During gameplay, the client sends the following commands to the server unreliably:

- move forward, up, sideways
- change orientation
- button presses
- impulse
- time between this command and the last

Each packet from the client contains the last three commands to compensate for lost packets.

A number of other non-gameplay messages can either be generated automatically by the client or by the user. These include instructions to spawn a new entity, kill an unruly player, pause the game, download a new map, change user information, etc.

1.2.7 Server to Client Messages

There's a are a large number of server to client messages. The ones that are directly involved in the gameplay and require low latency describe the state of each client in the game. Key elements of a client's state include that client's position, orientation, health, weapons, and ammo. Client state information is computed as a delta off a client's baseline state. All server-to-client messages are sent unreliably.

Chapter 2

Mirrored-Server Architecture

2.1 Introduction

This chapter describes the tradeoffs of the two basic multiplayer game architectures, the Client-Server and Peer-to-Peer (P2P) architectures, and proposes a new architecture, the Mirrored-Server architecture. The architectures vary in their end-to-end latencies, bandwidth requirements, degrees of administrative control and consistency protocol requirements. The Client-Server architecture is popular in the gaming industry because it allows companies to retain administrative control over servers and because no elaborate protocol is required to maintain consistency between different copies of the game state. The P2P architecture exhibits lower end-to-end latency and less concentrated bandwidth consumption. The Mirrored-Server architecture can achieve low latencies while still allowing administrative control of servers.

2.2 Architecture Goals

The choice of architecture depends on the characteristics of the game in question. The games considered in this report require very low latencies and a high degree of consistency over a complicated game state. First-person shooters such as Quake match this description closely. Many games do not typically fall into this category, including strategy games, role playing games, racing games, flight simulators, and sports games. Of secondary importance are minimizing bandwidth usage and maintaining administrative control over servers.

We consider games that are extremely sensitive to latencies. These games must be playable with latencies equal to the network latency, but the playability must degrade rapidly as the latency is increased. For a racing game, a study has found that latencies of 100 ms are almost unnoticeable, but as latencies reach 200 ms, the players have trouble controlling their race cars [15]. We have experienced similar latency sensitivity despite Quake. In contrast, many if not most games can function adequately with high latencies. In strategy and role-playing games, reaction times are not a factor. Even games that feel like action games may mask latencies by implementing vague or sluggish actions. For example, Everquest players report realistic fighting, even though Everquest uses TCP connections over the Internet [5]. Games can fake real-time interactions by requiring a second to sluggishly swing an axe, or by using weapons like spells where the exact time or direction of the action is unimportant or the results are difficult to verify.

The second criterion for relevant games is that they have strict consistency requirements over a

complex game state. Quake's game state must quickly and accurately reflect a number of permanent changes such as rocket fires, rocket impacts, damage and death. For racing games [15] and simple games like MiMaze [8], consistency requirements are much simpler. Two game states need only agree on positional information, which is refreshed rapidly, can be transiently incorrect and can be dead reckoned.

Bandwidth requirements for a multiplayer game can be significant. The Quake clients send out tiny command packets, and the Quake server performs delta updates and interest management to reduce state update sizes. Still, as you will see from our results, an eight-player game consumes 50 Kbps on the client side link, making modem play difficult. Even worse, the server consumes 400 Kbps, which means it must have good connectivity. If possible, we would like our architecture to reduce bandwidth requirements, particularly on the client-side links.

The fourth goal is to use an architecture that allows game companies to retain administrative control over servers. Administrative control can allow the game company to:

- Create persistent virtual worlds. The game company's servers can securely store player profiles and maintain centralized worlds that are always "on". For example, Everquest [5].

- Reduce cheating. If a secure server manages game state, opportunities for cheating may be reduced [1].

- Facilitate server and client maintenance. The game company can easily distribute bug fixes, and can allow the game to evolve after its release date.

- Eliminate piracy. Players can be forced to log in to verify their identity. [2]

- Enable player tracking. Everquest charges users based on the amount of time they spend playing.

The drawback of administrative control is that the game company is forced to run its own servers. Particularly for small companies, the cost of operating servers may be prohibitive.

2.3 Assumptions

For the following discussion, we assume that the gaming company operates a private low-latency network where mirrors are gateways to that network. This assumption makes the real-time multiplayer game problem tractable; the Internet cannot provide low latencies, nor can it handle gaming traffic that lacks congestion control mechanisms. A side benefit of this assumption is that we can choose to enable IP-Multicast within the private network, whereas IP Multicast is not available on the Internet. Without IP-Multicast, our architecture would require an end-host multicast protocol such as BTP [9], HyperCast [13] or NARADA [20]. Unfortunately, these end-host multicast protocols fail to provide the low latency requirements needed by our architecture.

2.4 Client-Server vs Peer-to-Peer Architecture

In this section, we will briefly describe the key features of the Client-Server and P2P architectures and compare them along the axes described in the previous section, namely: latency, consistency, bandwidth, and administrative control.

Most commercial games use a Client-Server architecture, where a single copy of the game state is computed at the server. This architecture is used in Quake [16], Starcraft [3] . The Client-Server architecture as used in Quake is depicted in Figure

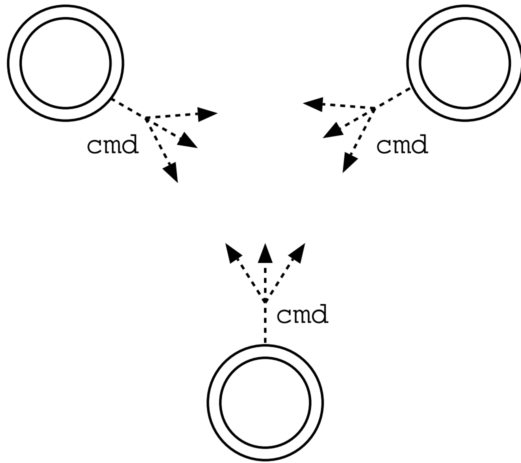


Figure 2.1: Peer-To-Peer Architecture

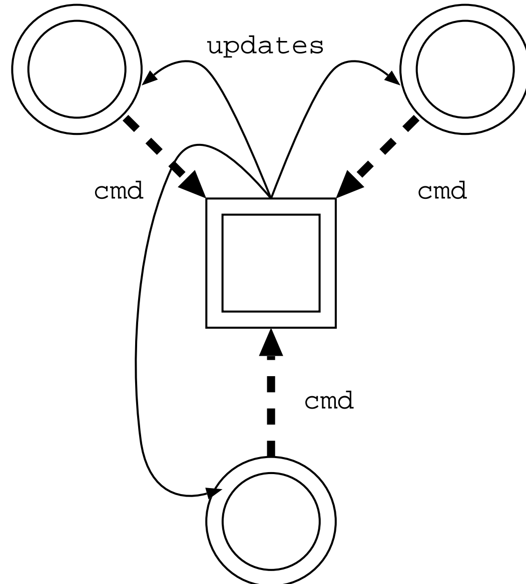


Figure 2.2: Client-Server Architecture

2.2. Clients input key presses from the user and send them to the server. We refer to these client-to-server messages as commands. The server collects commands from all of the clients, computes the new game state, and sends state updates to the clients. Finally, the clients render the new state. Notice that this is a thin client architecture, where clients are only responsible for inputting commands and rendering game state.

In the peer-to-peer (P2P) architecture, there is no central repository of the game state. Instead, each client maintains its own copy of the game state based on messages from all the other clients (see Figure 2.1). A P2P architecture has been used in a few simple games, including Xpilot [19] and MiMaze [8]. It is also used in complex, scalable battlefield simulators such as STOW-E. In MiMaze, clients multicast their game positions to every other client. Each client computes its own copy of the game state based on messages from other clients and renders that game state.

The primary advantage of the P2P architecture is reduced message latency. In the Client-Server architecture, messages travel from clients to a potentially distant server, then the resulting game state updates propagate back from the server to the clients. In the P2P architecture, messages travel directly from one client to another through the multicast tree. Because no widespread multicast service that we know of is provided on the Internet, in practice, latency-sensitive P2P messages need to be sent using unicast, which scales poorly with the number of clients.

In the Client-Server architecture, game state consistency is not a problem. There is only one copy of the game state, and that copy is maintained by a single authoritative server. In the P2P architecture, there are multiple copies of the game state. If messages are lost or arrive at different times, inconsistencies can arise. Although synchronization mechanisms such as our novel trailing state synchronization can resolve these inconsistencies, these mechanisms tend to increase the system's complexity and may require disorienting rollbacks.

The relative bandwidth consumption of the two architectures are roughly equivalent. In the P2P architecture, each host multicasts an input message every time period. This results in reception of N^2 total input messages per time period, where N is the number of hosts/players. In the Client-Server architecture, each host unicasts an input message to the central server every time period. Also, the server sends a game state message per client per time period. So the Client-Server architecture totals N input messages and N game state messages per time period. Typically, game state messages grow linearly with N because they must describe the change in state that occurred as a result of N input messages. Both architectures consume total bandwidth proportional to N^2. However, note that the traffic in the Client-Server architecture is concentrated at the central server, which must handle all N^2 messages.

Interest management techniques can be used to reduce the bandwidth requirements of both architectures [14]. Hosts are only interested in a small portion of the game state. If this portion of the game state is constant despite the addition of new players to the game, interest management will improve the scalability of both architectures. A centralized server can determine which pieces of state a client is interested in and only send them this filtered state information. The result, constant-sized state updates, would result in cN total bandwidth consumption. In a P2P architecture, a host can subscribe only to the multicast groups that match its area of interest. Again, the result would be cN total bandwidth consumption. A detailed discussion of interest management is beyond the scope of this paper. Overall, the relative bandwidth consumption depends on the sizes of different messages and the effectiveness of interest management techniques. Later we will present simple experiments that show that the total bandwidth usage of the two architectures is comparable.

The fourth axis along which we compare the two architectures is administrative control. Administrative control is trivial for Client-Server architectures. The game company need only run its own servers, where it can protect sensitive information such as player profiles and the server code itself. In the P2P architecture, most if not all of the code and data that are necessary to run a game must be resident on the clients, where they are vulnerable to hackers. Some administrative control can be retained by using a Client-Server protocol for non-gameplay messaging, but this does not resolve the security issues.

For most large scale massively-multiplayer games, gaming companies overwhelmingly choose Client-Server architectures, but DOD simulations have gone with the P2P architecture. The simplicity of dealing with a single repository of consistent state far outweighs the cost of increased latency. In addition, game companies may prefer to retain administrative control of their servers. On the other hand, other large organizations operate highly scalable distributed simulations. They generally have no administrative control problems because they control all of the clients.

2.5 Mirrored-Server Architecture

Our chosen architecture is the Mirrored-Server architecture, where game server mirrors are topologically distributed across the Internet and clients connect to the closest mirror. The Mirrored-Server architecture is a compromise between the two previously discussed architectures. As in the Client-Server architecture, the Mirrored-Servers can be placed under a central administrative control. Like the P2P architecture, messages do not have to pay the latency cost of traveling to a centralized server. Unfortunately, a Mirrored-Server system must still cope with multiple copies of the game state. We hope to solve this last remaining problem with our trailing state syn-

chronization mechanism.

In our mirrored version of Quake, clients send commands to their closest mirror. The mirror closest to the sender is referred to as the ingress mirror. The ingress mirror multicasts incoming commands to the other mirrors, which we term egress mirrors. The egress mirror calculates its own copy of the game state and sends state updates to its directly connected clients. The client then renders this updated game state.

Comparing bandwidth consumption among the three architectures becomes more complicated when assuming a high-speed network with each client sending messages to the nearest gateway. A detailed analysis of the bandwidth requirements of each architecture is presented in Section **??** The Mirrored-Server architecture uses more bandwidth on the private network where bandwidth is plentiful than it does on the public network. Also, the traffic concentration seen in the Client-Server architecture is shared by the mirrors in the Mirrored-Server architecture. Overall, the bandwidth consumption of all three architectures is comparable.

The Mirrored-Server architecture appears to be a reasonable alternative to the Client-Server architecture. Some of the larger gaming companies have already distributed game server clusters across the world. Both Blizzard [3] and Verant [5] have a number of geographically distributed clusters. To our knowledge, no game state is shared between these clusters.

2.6 Experiments and Results

The experiments in this section are designed to investigate the bandwidth requirements of the three architectures. We begin by developing a model to predict bandwidth usage based on the number of players in a game. Then, we collect some bandwidth statistics to populate the model. Our findings confirm that the architectures use similar amounts of bandwidth. The measurements are also used to evaluate the effectiveness of Quake's interest management techniques, but the results are inconclusive. We also include plans for future experiments in this section.

The bandwidth requirements of Quake can be described using a simple model. Clients always generate commands at a rate a, regardless of the architecture. The command streams increase to a rate b upon being forwarded to the multicast channel due to added timestamps and multicast headers. In both the Client-Server and Mirrored-Server architectures, the server sends state updates to each client at a rate $c + n(d + \alpha e)$. For the Mirrored-Server architecture, we assume 8 clients per server, $m = \frac{n}{8}$. Variable definitions are listed in Table 2.6.

First we calculate the asymptotic bandwidth requirements of the architectures as the number of clients increases. In Quake, interest management techniques remove only about 70% of the information about clients that are not in the area of interest, so they do not affect the asymptotic results. We further simplify the model by assuming that all messages are unicast. The asymptotic input and output rates of the clients, servers and mirrors are listed in Table 2.6. For the Mirrored-Server architecture, the server rates are for messages between the client and mirrors and the mirror rates are for messages between mirrors. Notice that all of the approaches use $\Theta(n)$ $\Theta(n^2)$ bandwidth on the public and private networks and bandwidth at the client. However, the Mirrored-Server architecture uses only $\Theta(n)$ bandwidth at the mirrors, while the Client-Server architecture uses $\Theta(n^2)$ bandwidth at the server.

To determine the values of a, b, c, d, e and α we set up a simple series of experiments. We ran a total of five experiments with $n = 1, 2, 3, 4,$ and 5 clients. For each experiment, the clients connected to a single mirror and played a two-minute game, 60 sec-

a	Command rate on the Client-Server connection
b	Command rate on the mirror-mirror connection
c	Rate of state updates with zero clients
d	Rate of information that is always included about each client
e	Rate of information about clients in proximity to the receiving client
α	Describes the effectiveness of the interest management techniques
n	Number of clients in the game
m	Number of mirrors

Table 2.1: Variable Definitions

	Client-Server	Peer-to-Peer	Mirrored-Server
Client Output Rate	a	bn	a
Server Input Rate	an		a
Client Input Rate	$c + (d + e)n$	bn	$c + (d + e)n$
Server Output Rate	$cn + (d + e)n^2$		$c + (d + e)n$
Mirror Output Rate			bn
Mirror Input Rate			bn
Private Network Rate	$an + cn + (d + e)n^2$	bn^2	bn^2
Public Network Rate	$an + cn + (d + e)n^2$	bn^2	$an + cn + (d + e)n^2$
Total Client	$a + c + (d + e)n$	bn	$a + c + (d + e)n$
Total Mirror/Server	$an + cn + (d + e)n^2$		$bn + c + (d + e)n$

Table 2.2: Asymptotic Input and Output Rates

a	5.68 Kbps
b	10.6 Kbps
c	1.16 Kbps
$d + e$	5.36 Kpbs

Table 2.3: Linear Regression Results

onds of which was used for analysis. The server was instrumented to record the transport-layer bandwidth consumption of client-to-server, server-to-client, and mirror-to-mirror messages. The coefficients a and b were directly observable. The values of c and $d + e$ were calculated by doing a linear regression on the server-to-client bandwidth measurements. Table 2.3 contains the measured coefficient values. The linear regression had a fairly encouraging correlation of 0.8682.

The asymptotic bandwidth usages of the architectures without interest management can be computed using the values of a, b, c, and $d + e$. First, Figure 2.3 compares our actual and predicted bandwidth usages. The actual server output rates are much lower than the predicted server output rates, although both increase quadratically. This discrepancy is due to Quake's interest management techniques, which are not reflected in our simple model. Table 2.4 show the bandwidth consumptions of each architecture. With Quake-like bandwidth configurations, the P2P architecture would use twice as much total bandwidth as the traditional Client-Server architecture. However, Quake's messaging protocols have been optimized for the Client-Server architecture, so this comparison is not fair. Note that the Mirrored-Server architecture uses a twice as much bandwidth on the private network where congestion is not a problem. The advantages of the Mirrored-Server architecture are illustrated in Figure 2.4, which compares the Client-Server server load and the Mirrored-Server mirror load. With four mirrors and 32 clients, mirrors consume 2 Mbs, while servers in the Client-Server architecture must handle 6 Mbs.

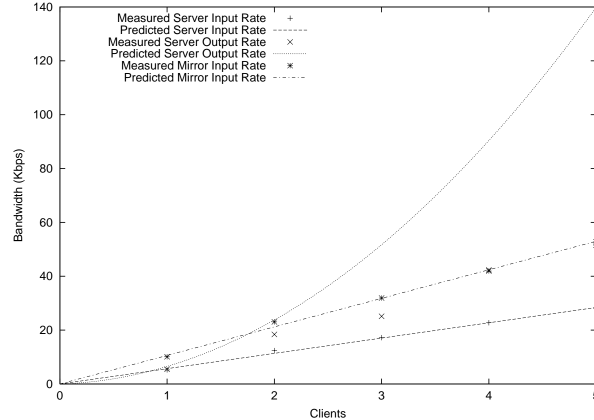


Figure 2.3: Measured vs. Predicted Bandwidth Usage

To determine the success of Quake's interest management techniques, we attempted to estimate the values of c, d and e separately. Given these values, α can be computed for each trace. Remember that α represents the proportion of the other clients whose detailed state is included in any given state update. Because d and e are both coefficients of n, it is impossible to separate them using regression analysis. Instead, we looked at the source code and attempted to estimate how many bytes per state update would be devoted to c, d and e. These estimates are imperfect because the amount of data written for each client depends on how much that client's state has changed since the last state update.

Table 2.5 lists the value of α for each set of measurements. We expected a fixed percentage of the other clients to be represented in any state update regardless of the number of clients. Unfortunately, the results are not strongly correlated. We need to perform more experiments with a larger number of clients to better understand the behavior of Quake's

	Client-Server	Peer-to-Peer	Mirrored-Server
Private Network Rate	$5.36n^2 + 6.84n$	$10.6n^2$	$10.6n^2$
Public Network Rate	$5.36n^2 + 6.84n$	$10.6n^2$	$5.36n^2 + 6.84n$
Total Client	$5.36n + 6.84$	$10.6n$	$5.36n + 6.84$
Total Mirror/Server	$5.36n^2 + 6.84n$		$53.5n + 9.28$, $\frac{n}{m} = 8$

Table 2.4: Bandwidth Consumption

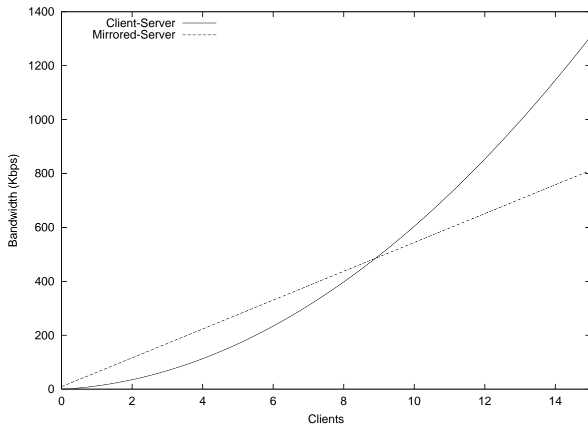


Figure 2.4: Predicted Client-Server and Mirrored-Server Bandwidth Requirements

Clients	Clients in State Updates (%)
1	N/A
2	66.7
3	13.2
4	28.5
5	16.7

Table 2.5: α Values

interest management mechanism. Also, better instrumentation could be installed to eliminate the error due to rough estimations of the coefficients c, d and e. We reserve this for future work.

Latency is the most important experimentally verifiable difference between the architectures presented in this section. Unfortunately, coming up with meaningful latency comparisons involves choosing a realistic network topology. At one extreme, a star topology centered at the server will result in identical latency for both the Client-Server and P2P architecture (see Figure 2.5a.). At the other extreme a topology where the server is located in an infinitely distant or congested part of the network, as in Figure 2.5b., will result in the P2P architecture being infinitely faster than the Client-Server architecture. For the Mirrored-Server architecture, the best and worst cases are less clear. The topology in Figure 2.5c. will result in much better average end-to-end latency for the Mirrored-Server and P2P architectures, because a centralized server would have to be placed far away from half of the clients. The topology-dependent study evaluation is reserved for future work.

2.7 Conclusions

Each of the three architectures sports a different set of advantages and disadvantages. No single architecture is suited to all classes of multiplayer games. The Client-Server architecture has poor latency and good consistency characteristics, so it is a good match for any non-real-time game. The P2P architecture has good latency but poor administrative control char-

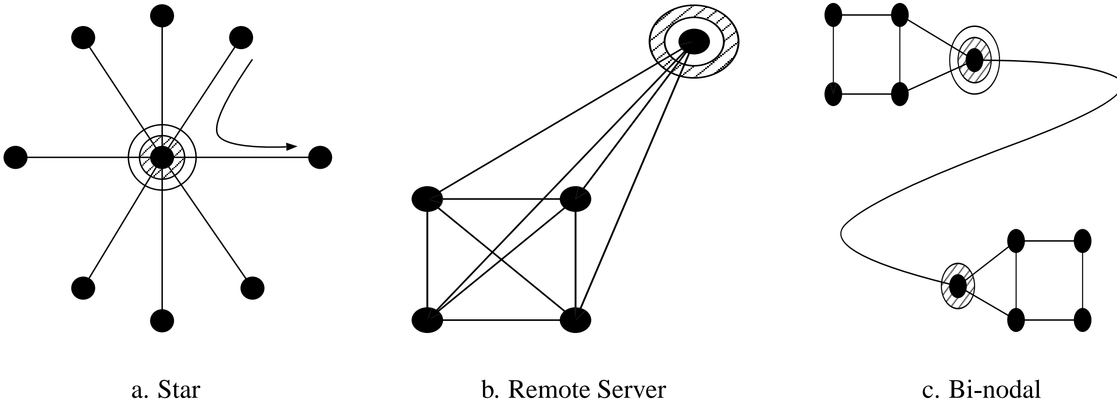


Figure 2.5: Different topology types

acteristics, making it ideal for small scale, latency-sensitive games. The Mirrored-Server architecture achieves low latency and administrative control; we feel it is a great solution for next-generation real-time persistent-world games.

Chapter 3

Consistency Protocols and Synchronization Mechanisms

3.1 Introduction

The Mirrored-Server and P2P architectures involve computing multiple copies of the game state. Lost or delayed messages can cause inconsistencies between game states. A consistency protocol is needed to deliver messages to their destinations, and a synchronization mechanism must be used to detect and resolve inconsistencies. Although the Mirrored-Server and P2P architectures pose identical consistency problems, we will focus on the Mirrored-Server architecture because it is the chosen architecture for our Quake system.

This chapter begins with some key definitions and an overview of our consistency goals. Next, we launch into a description of two consistency protocols. The last half of the chapter concerns our synchronization mechanism, trailing state synchronization (TSS). We set the stage with a discussion of previously proposed synchronization mechanisms and how our approach differs from those mechanisms. TSS is described in detail, with some analysis of optimal configurations. The results in this section support our claim that TSS is an efficient, low-latency, consistent synchronization protocol that is appropriate for multiplayer games.

3.2 Definitions

We make a distinction between the consistency protocol and the synchronization mechanism. The *consistency protocol* is responsible for delivering messages to the destination mirror and incorporating those messages into the game state. A key piece of the consistency protocol is the *synchronization mechanism*, which detects and resolves inconsistencies. The consistency protocol gives a high-level specification of the messaging protocols, while the synchronization mechanism deals with the details of maintaining consistent state.

Commands are simple messages that convey input from a client to a server. In Quake, commands contain key presses and orientation information. Quake servers calculate the game state directly from commands. If instead we think of the Quake server as a simulator, then it is responding to *events* by changing the game state. We have simplified the Quake gameplay so that the following events are sufficient to initiate all game state changes.

- Move event: An avatar moves to a new position.
- Fire event: An avatar fires a rocket.
- Impact event: The rocket impacts and detonates

- Damage event: An avatar takes damage and possibly dies.
- Spawn event: An avatar is reborn in a random part of the map.

The clients generate move, fire and spawn events, while the mirrors generate impact and damage during the game state computation.

A key observation is that different events have different consistency and latency requirements. *Real-time events* have strict timeliness and lax consistency requirements. At the other end of the spectrum, *consistent events* have lax timeliness and strict consistency requirements. The most challenging events are *consistent real-time events*, which have strict timeliness and strict consistency requirements.

Real-time events are those events that players must be informed of in real-time in order to react reflexively. The best example of a real-time event is the move event. Any delay in the reaction time of one's own avatar movement inhibits the effectiveness of player reflexes. Delays in the perceived motion of other avatars force the player to anticipate their motion. Because shooting another player requires significant hand-eye coordination, players are extremely sensitive to the timeliness of move events. If possible, real-time event latency must be less than 100 ms. Although human reflexes can act on the order of 100 ms, players cannot easily detect short-lived inaccuracies in avatar position. Therefore, real-time events can occasionally be lost or delayed without ill effect.

All events that take more than 100 ms to react to are termed consistent events. Damage events, die events, and spawn events can all be placed into this category. Unlike real-time events, these events affect the game state in permanent and noticeable ways. Therefore, consistent events must be consistently processed across all clients. For example, all clients must agree that an avatar took a hit and died. Here, consistency is far more important than timeliness, so we might be willing to wait 500 ms before the avatar keeled over.

Unfortunately, there are some events that have tight real-time constraints and require a high degree of consistency. Fire events and impact events are two examples. Clearly, all clients must agree on whether or not a rocket hit an avatar or whether that avatar dodged, allowing the rocket to whiz by. Assume that we treated this event like the consistent events, where there could be a 500 ms delay after the event occurred before the event is reflected in the game state. Then all weapon impacts would occur after they have struck their targets. This means that rockets would continue along their trajectory for a fraction of a second, possibly passing through their target before detonating. This behavior is unacceptable, so we must treat consistent real-time events differently.

3.3 Consistency Protocols

3.3.1 Consistency Protocol Goals

The goals of the consistency protocol follow directly from the event class definitions. Real-time events must appear in the game state with the lowest possible latency and consistent events must be consistent between game states. The mechanisms to accomplish this should be as efficient as possible, both in bandwidth and computational load. For example, if real-time event servicing can be made more efficient than consistent real-time event servicing, the same mechanism should not be used for both.

3.3.2 Distributing Game State Computations

So far, we have not explicitly examined the issue of where the game state calculation should occur.

For this discussion, it helps to think of each ingress mirror as a sender in the peer-to-peer architecture, and the egress mirrors as the receivers. Should the ingress mirror calculate the new game state and then send a state update to the egress mirror? Or should the ingress mirror just forward the incoming command to the egress mirror and allow it to generate the new game state? As a compromise, should the ingress mirror perform some computation, send an intermediate result to the egress mirror, and then allow the egress mirror to complete the computation?

Computing the game state at the ingress mirror is a strange proposition. This has the advantage that the game state calculation need occur only once. However, if different pieces of game state are computed at different ingress mirrors without input from other mirrors, there will be no interactivity between clients on different mirrors. For example, two ingress mirrors could simultaneous decide that two different avatars occupy the same position on the map instead of coordinating to detect a collision.

The more natural solution is to have all game state calculated at the egress mirror. This approach is used in the command-based consistency protocol. In this configuration, the ingress mirror forwards commands to the egress mirrors, and each egress mirror computes its own copy of the game state. This approach requires redundant calculations, but collisions and other interactions are handled properly because the egress mirror makes decisions based on input from all the clients.

A compromise between the two approaches is to compute game state that is entirely under the control of the ingress mirror at the ingress mirror. For example, the ingress mirror could take raw front-end commands and convert the into different events (see Section 3.4). This is the key feature of the event-based consistency protocol. It could also perform initial avatar position calculations, which would then be modified by the egress mirror to account for dynamic collisions. This is the approach used in MiMaze, which does an initial position calculation at the ingress mirror, then calculates the final interactions at the egress mirror [8].

The decision on where to place the game state computation may be based on which representation consumes the least bandwidth. In Quake, the initial command is very small. After some event calculation, the generated events might be slightly larger than a command. If a significant amount of game state were calculated at the ingress mirror, the resulting game state updates could be huge.

In our implementation, we elected to use the command-based consistency protocol, where no game state computations are performed at the ingress mirror. This decision produced a much simpler implementation where most of the Quake code remained intact. Dividing the game state computation between the egress mirror and the ingress mirror would have required a complete rewrite of the server code into modular event handlers. In general, keeping all of the game state computation on one server becomes more attractive as the game becomes more complicated. Although we chose the command-based consistency protocol, we also investigate the event-based consistency protocol.

3.3.3 Command-Based Consistency Protocol

Our implementation places all of the game state computation at the egress mirror. The ingress mirror timestamps commands from the client and reliably multicasts them on to the egress mirror. Given the commands from each client and the time at which they were issued, the egress mirror can compute the correct game state. The egress mirror sends game state updates only to its directly connected clients. The command-based consistency protocol is simple and provides low latency, reliable delivery of

all commands. However, it does not distinguish between different classes of events and requires a many-to-many, low-latency, reliable multicast protocol.

By using a reliable multicast protocol, we ensure that all commands eventually reach the egress mirror. The egress server can compute the correct game state by executing the commands in the order in which they were issued. Because we perform optimistic execution of command packets, we must correct the game state when commands are received out of order. These corrections, or rollbacks, can cause unexpected jumps in the game state, severely degrading the user experience. Therefore, we must at all costs minimize the frequency and magnitude of rollbacks. Informally, we have found that rollbacks of less than 100 ms are unnoticeable. The details of the trailing state synchronization mechanism are presented in section 3.4.

The main feature of the command-based consistency protocol is simplicity. Our command-based protocol requires minimal modifications to the server when the protocol is added to an existing game. To the existing server back-end running on the egress mirror, commands from any of the ingress mirrors can appear to have come from a client attached directly to the local mirror. On the ingress mirror the changes are minimal as well. Instead of calling the local back-end to execute a command after pulling it up from the client, it multicasts it to the other mirrors, and then later processes it like every other command. Other consistency protocols require at least partial processing at the ingress mirror, and changes to the back-end to accept partially processed commands instead of raw commands. When a game is not designed with distribution in mind this is an important feature.

Aside from simplicity, the command-based consistency protocol has several secondary advantages. First, it achieves consistency without the need for a third-party trusted server. Second, it should result in reasonably efficient bandwidth usage by using multicast and sending only small command packets. Third, it will achieve the lowest possible latency because commands are sent directly from mirror to mirror across a private network. The weakness of this protocol is that it requires low latency many-to-many multicast, which is a difficult problem. Our reliable multicast protocol, CRIMP, is covered in Chapter 4.

3.3.4 Event-Based Consistency Protocol

Previously, we stated that the ingress mirror can process incoming commands in order to generate different events. Real-time events, consistent events, and consistent real-time events can then each be handled in a different way. In this section, we present a consistency protocol that does not require low-latency many-to-many multicast. Instead, it relies on an überserver and a standard (not low latency) one-to-many reliable multicast protocol.

In this protocol, a single überserver is responsible for maintaining the authoritative game state, while each egress mirror maintains a local, possibly inconsistent game state. The überserver is responsible for informing the egress mirrors of consistent events. If an egress mirror disagrees with the consistent event, it must perform a rollback to the correct state.

The ingress mirror generates move events in response to move commands. Remember that move events require low latency but not reliability. Therefore, move events can be multicast to other egress mirrors using unreliable multicast. However, in order to ensure approximately correct game state at the überserver, move events are unicast reliably from the ingress mirror to the überserver. There are no consistency checks for move events.

Consistent events are generated solely by the überserver, because the überserver is the only server with the correct positional information. For example,

the überserver is the only one that can determine exactly how much damage an avatar took because it is the only server that knows exactly how far an avatar was from the blast. The überserver sends consistent events to the egress mirrors using reliable multicast. On reception of a consistent event, the egress mirror makes the necessary changes in the game state. Note that the reliable multicast protocol could take on the order of a second to deliver the consistent event.

The most challenging events are consistent real-time events. The only way to deal with such events is to process them optimistically, then do a rollback if the event was processed out of order. A mechanism for performing efficient rollbacks is presented in the next section 3.4. The ingress mirror generates fire events in response to fire commands from the client. However, the ingress mirror does not know the exact position and orientation of the shooter, so it cannot generate an authoritative fire event. The non-authoritative fire event is multicast unreliably to the egress mirrors and processed as if it were the correct fire event. It is also forwarded reliably to the überserver, which generates an authoritative fire event. This authoritative fire event is multicast reliably to the egress mirrors. If the authoritative and non-authoritative fire events differ, the egress mirror performs a rollback using the trailing state synchronization mechanism.

Although the event-based consistency protocol is probably a superior solution, it is much more complicated than the command-based consistency protocol. Implementing the event-based consistency protocol would have meant ripping apart the very poorly written Quake code into event handlers. We must leave this ordeal for someone with more time on their hands.

3.4 Synchronization Mechanisms

In order for each mirror to have a consistent view of the game state, we need some mechanism to guarantee a global ordering of events. This can either be done by preventing misordering outright, or by having mechanisms in place to detect and correct misorderings. At the same time, if we are unable to maintain ordering within a reasonable delay, no one will be willing to use the game servers. In both military simulations and simpler games, similar problems exist and there are a number of synchronization algorithms used in these cases [7]. However, none of these algorithms work well in a fast-paced multiplayer game such as Quake. Unlike other uses of synchronization, the events in Quake are not directly generated by other events – they are generated by user commands. Therefore, many of the concepts in these other algorithms such as event horizons, fixed buckets and rounds of messages are not applicable. Likewise, unlike less complicated games, it is not sufficient to be mostly consistent and dead reckon the rest. The speed at which game state changes causes these types of errors to quickly multiply and lead to large divergences.

3.4.1 Conservative Algorithms

Lock-step simulation-time driven synchronization [7] is by far the simplest, and by far the least suitable for a real-time game. No member is allowed to advance its simulation clock until all other members have acknowledged that they are done with computation for the current time period. This takes the first approach to preventing inconsistencies from possibly being generated. In this system, it is impossible for inconsistencies to occur since no member performs calculations until it is sure it has the exact same information as everyone else. Unfortunately, this scheme also means that it is impossible to guar-

antee any relationship with real wall-clock time. In other words, the simulation does not advance in a clocked manner, much less in real time. In an interactive situation such as a multi-player game, this is not an acceptable tradeoff. There are a number of similar algorithms and variants such as fixed time-bucket synchronization which still suffer from the same problem of simulation-time and wall-clock having no real correlation.

Chandy-Misra synchronization does not require the coordinated incrementing of simulation time like the previous two algorithms. Instead, each member is allowed to advance as soon as it has heard from every other member it is interacting with. Additionally, it requires that messages from each client arrive in order. This requires the concept of a round of messages to be of any use. In a mirrored game server architecture, different servers will have different event generation rates depending on the number of connected clients. This scheme limits execution to the rate of the slowest member. Additionally, the times at which these events are generated will have little relation with the generation of earlier events.

3.4.2 Optimistic Algorithms

The two types of algorithms above have taken a cautious approach to synchronization. There are also several algorithms which execute events optimistically before they know for sure that no earlier events could arrive, and then repair inconsistencies if they are wrong. These types of algorithms are far better suited for interactive situations.

Time Warp synchronization works by taking a snapshot of the state at each execution, and rolling back to an earlier state if an event earlier than the last executed event is ever received. Periodically, all members reset the oldest time at which an event can be outstanding, thereby limiting the number of snapshots needed. On a rollback, the state is first restored to the snapshot, and then all events between the snapshot and the execution time are re-executed. Additionally, since, like the other algorithms, Time Warp assumes that events directly generate new events, as part of the rollback anti-messages are sent out to cancel any now invalid events (which in turn trigger other rollbacks). The big problem with Time Warp in a game such as Quake is the requirement to checkpoint at every message. A Quake context consumes about one megabyte of memory, and new messages arrive at a rate of one every thirty milliseconds for each client. Additionally, copying a context involves not just the memory copy but also repairing linked lists and other dynamic structures. The other problem of Time Warp, anti-message explosion, is not as important since the Mirrored-Servers do not directly generate new events in response.

There are additionally a class of algorithms which are "breathing" variations on the above algorithms. Instead of fully optimistic execution, breathing algorithms limit their optimism to events within an *event horizon*. Events beyond the horizon can not be guaranteed to be consistent, and are therefore not executed. Since in Quake events do not directly generate new events, this concept does not work.

The algorithm implemented in MiMaze [8] is an optimistic version of bucket synchronization. Events are delayed before being executed for a time which should be long enough to prevent misorderings. If events are lost or delayed however, MiMaze does not detect an inconsistency and attempt to recover in any way. If no events from a member are available at a particular bucket, the previous bucket's event is dead reckoned; if multiple events are available, only the most recent is used. Late events are just scheduled for the next available bucket, but, because only one event per bucket is executed, are not likely to be used. For a simple game such as MiMaze, these optimizations at the cost of consistency are claimed to be acceptable. For a much faster paced multi-

player game though, a higher level of consistency is required.

3.5 Trailing State Synchronization

As described above, none of the existing distributed game or military simulation synchronization algorithms are well suited to a game such as Quake. Our solution to this problem is trailing state synchronization (TSS). Similar to Time Warp, TSS is an optimistic algorithm, executing rollbacks when necessary. However, it does not suffer from the high memory and processor overheads of constant snapshots. TSS also borrows several ideas from MiMaze's bucket synchronization algorithm. Instead of executing events immediately, synchronization delays are added to allow events to reorder. The end result is an algorithm which has the strong consistency requirements needed by a rapidly changing game without the excessive overhead of numerous snapshots of game state. Because TSS was designed with our mirrored game server in mind, we refer to *commands* being synchronized as opposed to the *events* in the last section. This is merely semantic and has no bearing on the operation of TSS.

In order to recover from an inconsistency, there must be some way to "undo" any commands which should not have been executed. The easiest way to handle this problem is to "roll-back" the game state to when the inconsistency occurred, and then re-execute any following commands. The difficult part is where to get the state to roll back from. Since any command could be out of order and there are hundreds of commands generated every second, the number of states needed quickly grows out of hand. Even if a limit is placed on how old a command can be and still be successfully recovered (creating a window of synchronization), to maintain reasonable consistency this limit will probably still be larger than the number of snapshots we would like to keep. Instead of keeping snapshots at every command, TSS keeps multiple constantly updating copies of the same game, each at a different simulation time. If the leading state (which is the one that actually communicates to clients) is executing a command at simulation time t, then the first trailing state will be executing commands up to simulation time $t - d_1$, the second trailing state commands up to $t - d_2$ and so on. This way, only one snapshot's worth of memory is required for each trailing state, reducing and bounding the memory requirements.

TSS is able to provide consistency because each trailing state will see fewer misordered commands than the one before it. The leading state executes with little or no synchronization delay d_0. The synchronization delay is defined as the difference between wall-clock time and simulation time, used to allow commands to be reordered properly at the synchronizers before execution. If a command is stamped with wall-clock time a at the ingress mirror, then it cannot be executed until wall-clock time $a + d$, even on the same ingress mirror. With a delay of zero, the leading state will provide the fastest updates to it's clients (preserving the "twitch" nature of the game) but also very frequently be incorrect in it's execution. The first trailing state will wait longer before executing commands, and therefore will be less likely to create inconsistencies but also less responsive to clients. This continues, forming a chain of synchronizers each with a slightly longer synchronization delay than its predecessor.

In order to detect inconsistencies, each synchronizer looks at the changes in game state that the execution of a command produced, and compares this with the changes recorded at the directly preceding state. In Quake, there are two basic classes of events that a command can generate. The first type we refer to as *weakly consistent*, and consists of move commands. With these events, it is not essential that the

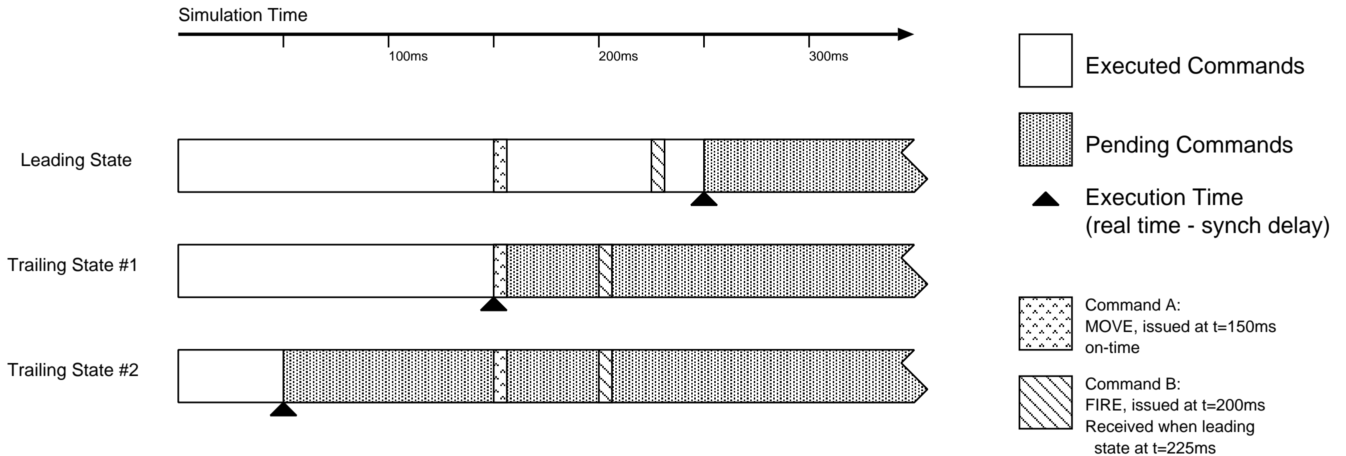


Figure 3.1: Trailing State Synchronization Execution

same move happened at the same time as much as that the position of the player in question is within some small margin of error in both states. The other class is *strictly consistent*, and for these events (such as missiles being fired) it is important that both states agree on exactly when and where it occurred.

If an inconsistency is discovered, a rollback from the trailing state to the leading state is preformed. This consists of copying the game state as well as adding any commands after the rollback time back to the leading state's pending list. The next time that state executes, it will calculate it's execution time as $t-d$ and execute all the commands again. The difference in delays between states determines how drastic the rollback will be. Additionally, the rollback of one state may cause, upon it's re-execution, inconsistencies to be detected in its leading state. In this fashion, any inconsistencies in a trailing state which the leading state also shares will be corrected. The last synchronizer has no trailing state to synchronize it, and therefore any inconsistencies there will go undetected. However, if it is assumed that the longest delay, as in the other bounded optimistic algorithms, is large compared to expected command transit delays, this is unlikely to occur.

3.5.1 An Example of TSS

Figures 3.1 and 3.2 depict a simple example of TSS. There are three states in the example with delays of 0ms, 100ms and 200ms each. Two commands are hi-lighted. Command A is a MOVE, issued (locally) at $t = 150$ and executed immediately in the leading state. At time $t = 250$, the first trailing state reaches a simulation time of 150 and executes command A. Since A was on time, it's execution matches the leading state's and no inconsistency occurs. Similarly, at time $t = 350$, the final trailing state reaches simulation time 150 and executes command A. It too finds no inconsistency, and no one is left to check it (this is a contrived example, in real life you would likely want a longer delay than 200ms on the last synchronizer). Command B is a FIRE event, issued at time $t = 200$ on a remote mirror. By the time it arrives, the real-time is $t = 225$. The command is executed immediately in the leading state and placed in the proper position in the other two states since they are

at simulation times 100 and 0. At time $t = 300$, the first trailing state executes B. When it compares it's results with the leading state's, it is unable to find a FIRE event from the same player at time 200, and signals the need for a rollback. Figure 3.2 zooms in on the recovery procedure. The state of the trailing state is copied to the leading state which places it at time 200. The leading state then marks all commands after time 200 as unexecuted and re-executes them up to the current t. This example hi-lights one of the features of TSS. It is possible that there were other inconsistencies in the gap between times 200 and 300 (a burst of congestion perhaps). The recovery of the first inconsistency at time 200 in effect canceled any future recoveries in this window.

3.5.2 Analysis of TSS

Although similar to many other synchronization algorithms, TSS has key differences with each of them. It is clearly very different from any of the conservative algorithms, since its scheduling of execution is based on synchronization delay and not when it is safe. It is also clearly different from MiMaze's bucket synchronization since it provides absolute synchronization for events delayed no later than the longest synchronization delay. MiMaze on the other hand does not really detect let alone recover from any inconsistencies it may cause. TSS and Time Warp both execute commands as soon as they arrive. They differ however in their methods of recovering from inconsistencies. TSS is a little more optimistic than Time Warp in that it does not keep a snapshot of the state before executing every command so that it can recover as soon as a late command arrives. Instead it catches inconsistencies by detecting when the leading state and the correct state diverge, and correcting at that point. It is possible, especially with weakly consistent events, that though executed out of order commands may not cause an inconsistency.

TSS will perform best in comparison to other synchronization algorithms when three situations are present: the game state is large and expensive to snapshot, the gap between states' delays is small and easy to repair, and event processing is easy. The first is definitely present in Quake, with one megabyte of data per context. The second is a parameter of TSS and therefore it is possible to tweak the number and delays of synchronizers (see Section 3.5.3). The cost of processing each command is not immediately apparent, and we must determine this experimentally. A preliminary study of the performance of TSS is in Section 3.6.

3.5.3 Choosing Synchronization Delays

Picking the correct number of synchronizers and the synchronization delay for each is critical to the optimal performance of TSS. If too few synchronizers are used, in order to provide a large enough window of synchronization the gaps between synchronizers must necessarily be large. This leads to greater delay before an inconsistency is detected, and more drastic and noticeable rollbacks. Conversely, if too many synchronizers are used the memory savings provided by TSS will be eliminated. Additionally, rollbacks will likely be more expensive since a longer cascading rollback is needed before reaching the trailing state. Given information on the network and server parameters, (delay distribution, message frequency, cost of executing a message and the cost of a context copy) it should be possible to build a model to calculate how many synchronizers should be used and with what delays. Due to time constraints, this aspect of TSS has not been fully explored.

3.5.4 Implementation

The first step in implementing TSS, or any optimistic synchronization algorithm for that matter, in

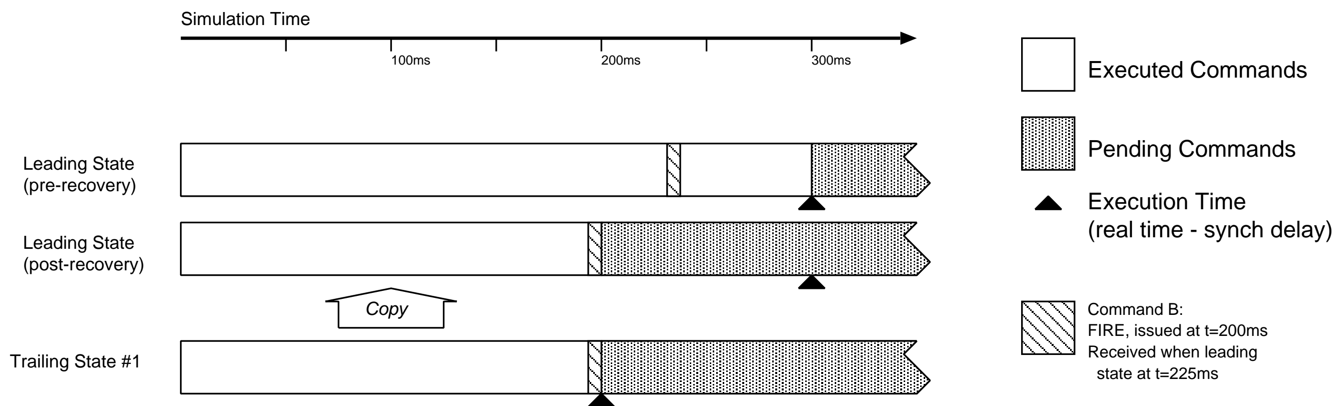


Figure 3.2: Trailing State Synchronization Rollback

Quake was to alter the server so that all game state data was within a context, and create functions capable of copying one context onto another. Although Quake uses almost no dynamic memory for performance reasons, they make numerous very ugly hacks to their static structures to avoid extraneous dereferencing of pointers for the same reason. Additionally, the actual game play within the server is not done in C code, but in a primitive interpreted byte code language known as QuakeC. It was because of the amount of work done within QuakeC that we decided to simplify the Quake game to a single weapon and level.

Once the Quake server had been altered to use contexts, it was fairly simple to add synchronization. Instead of executing client packets immediately, they are intercepted and sent out over the multicast channel. Upon receiving a multicast command, it is inserted back into Quake's network buffer and parsed by quakes normal function. All commands other than moves (which also include fires) get executed normally in all contexts, while the moves are inserted into the synchronizers. Every time through the main event loop, each of the synchronizers is checked, and any pending commands which can be are executed, and inconsistencies checked for.

In addition to the mirroring and synchronization, we also added a trace feature to the server. This logs to a file every command sent to the multicast channel and allows games to be replayed exactly in the future for deterministic simulations.

3.6 Experimental Results

To test the performance of TSS, we ran a series of simulations using the trace feature described above with different network and synchronization parameters. Unfortunately, due to QuakeC code still exhibiting randomness in certain situations, even with a single server and two states inconsistencies would occasionally occur. Because of this, a detailed and accurate study of the behavior of TSS with different configurations and different network conditions was not possible.

From the experiments we were still able to gather several useful results. The results in Table 3.6 show seven runs, all using the same trace file with three users connected to each of two servers. The statistics were gathered at mirror one, which saw 18593 commands from local clients and the multicast group.

Synchronization Delays	Executed Commands	Execution Time	Rollbacks	Rollback Time	Total Time	Command Cost	Rollback Cost
0,50	40780	6.135408	817	1.148895	8.953207	.15ms	1.41ms
0,100	45401	6.369374	870	1.226166	9.317506	.14ms	1.41ms
0,50,100	59981	9.024021	938	1.315154	12.296119	.15ms	1.40ms
0,100,1000	331687	26.195788	6687	10.105350	43.772133	.08ms	1.51ms
0,50,100,150	79357	12.144979	1092	1.534347	15.904039	.15ms	1.41ms
0,50,100,500	99730	13.261478	2370	3.361073	19.375112	.13ms	1.42ms
0,50,500,1000	251044	23.223288	6995	10.513721	39.266619	.09ms	1.50ms

Table 3.1: Trailing State Synchronization Execution Times

For each configuration of TSS, the seconds spent executing commands (including re-execution of commands due to rollbacks) and those spent performing rollbacks (copying contexts and moving commands back to the pending list) as well as the number of occurrences of these events is listed. These are then used to calculate a per-event cost of executing a command or performing a rollback. In all the cases these costs were nearly identical, with command execution being an order of magnitude less expensive. This supports the third condition for TSS to be advantageous, that event processing be inexpensive. For every command executed, TSS does an order of magnitude better than Time Warp or other similar optimistic algorithms.

3.7 Future Work

As discussed above, a more through examination of the parameter space for synchronization delays is needed. Additionally, taking the idea of weakly consistent events one step further, it would be interesting to look at the effect of correcting positional differences between states as opposed to performing a full rollback. While we were limited in what we could implement by Quake's existing design, in another situation the Event based consistency protocol might be implemented and studied. Finally, even though it is entirely subjective, a better user study to determine what delay and rollback rates are unnoticeable, what are noticeable but playable, and at what point the game becomes unplayable is needed to determine the true usefulness of the mirrored multi-player game servers.

Chapter 4

Reliable Multicast

4.1 Introduction

As stated in earlier sections, our architecture requires a reliable many-to-many delivery method between mirrors. In this chapter, we investigate the requirements of a delivery method, and provide a design, the Clocked Reliable Interactive Multicast Protocol *(CRIMP)*, that meets them.

This part is organized as follows. Section 4.2 describes the requirements and desired features of our delivery method. In Section 4.3 we discuss other designs in literature that fueled our design decisions. Section 4.4 provides a description of our architecture. In Section 4.6 we provide some evaluation and analysis of CRIMP. We close the chapter with future work.

4.2 Requirements and Simplifications

To provide perfect consistency, our architecture requires that all packets be delivered to each mirror. Each server provides consistency by executing the commands sent between mirrors. If packets are dropped, the servers' states could diverge, in essence, no longer being consistent.

The mirrored architecture, from a quality of play point of view, also requires that packets be delivered in a low latency manner. If a long delay synchronizer plays back a packet which an earlier synchronizer did not play back, the mirror is forced to roll-back. Our architecture evaluation shows why this circumstance should be avoided. If possible, a reliable delivery method designed for our architecture should minimize what we call *perceived latency*. We define perceived latency as the difference between when a packet is originally sent by the sender, t_s, and when a receiver receives the packet, t_r.

Since the consistency layer of the architecture does its own ordering of packets, packet ordering is not required by the delivery layer. Instead, the synchronizer insists that the packets be passed up to the application layer as soon as they arrive.

As described in [12], most packet loss is due to congestion, and the delivery layer must carefully react to packet loss. The delivery layer should avoid flooding the network when a loss is detected to avoid deteriorating network conditions further.

The delivery layer can use the abstraction of an überserver. This is an entity on the delivery channel which can provide authority on any issues in the channel.

Since the mirrors in the channel forward client commands to the many-to-many delivery channel, delivery rate is thus determined by the clients, and not the mirror. Congestion control must thus be done

with the cooperation of the clients. Since we are unable to modify the Quake client, congestion control is out of the scope of our design.

Since each mirror has a finite number of synchronizers, after a packet misses its playback point in the last synchronizer, it can no longer be used. Thus, each packet has a finite life in the system.

4.3 Discussion

The idea of reliable many-to-many communication is not a new concept. In this section, we will provide a discussion of which ideas we can borrow, and which we have to discount for our delivery layer.

4.3.1 Unicast

Naive Unicast

The most basic many-to-many method of providing reliability is a full mesh of connections between entities on the channel. Using this method, a sender has connections to each of the receivers. Either the sender or the receiver detects losses and prompts the sender to retransmit lost packets. This method may work well on a small scale, but as noted in [20], it causes unnecessarily high link usage, especially at the sender, as it must send indiviual messages to each receiver. In addition, all recovery messages must come from the sender, and thus time to recovery is bound by the longest RTT. Because of the problems this design, we discounted naive unicast as an infeasible method.

End-Host Multicast

Another method of providing a many-to-many reliable delivery is to use end-host multicast [20], whereby entities unicast messages to each other based on some sort of global topology. Reliability is enforced between neighbors in this method, generally in the transport layer. However, as the authors concede, latency is generally higher than other many-to-many methods such as IP-Multicast [20]. Packets must propagate through the topology until they reach every host. This is quite scalable, yet it most likely would not meet the latency requirements of our architecture.

4.3.2 IP-Multicast

To solve the problem of propagation delay of end-host multicast, we can use IP-Multicast to send efficiently to many hosts. However, since IP-Multicast is built at the IP-layer, to this date, no standard reliable transport protocol is available for it. However, research [11, 18, 4] has been done making IP-Multicast scalably reliable. For simplicity, we categorize these efforts into two separate categories, Topology-Based, and Receiver-Based.

Topology-Based Reliability

In this category of reliable IP-Multicast, receivers are grouped together in trees, rings or some other topology [11]. Reliability is ensured by having receivers aggregate their acknowledgments and sending them back to the sender, which retransmits. Local recovery is an enhancement of this, whereby some member of the aggregate group fill the requests for retransmission between themselves [11]. The authors of [18, 11] show this to be a very scalable approach to reliability over IP-Multicast. However, the scalability comes at the cost of latency, as retransmission must wait for NAKs to propagate up from a receiver through the topology, or use ACKs and RTO timers, which is bound by RTTs.

Receiver-Based Reliability

This category of reliability, as described in [4] uses receiver NACKs to fuel sender retransmission. The receivers detects losses and sends a recovery request back to the multicast group. Anyone who stored the packet can respond to the request with the data. One implementation of this, SRM, sets RTT based timers to ensure that the multicast channel is not flooded with requests. This method is less scalable than topology-based reliability, in that all entities must keep state of all other entities on the channel. However, the timers set ensure that network utilization is kept to a minimum. The authors of [12] show that in the ideal case, SRM provides the lowest latency recovery of known solutions. Observe that Topology-Based Reliability is actually a subset of Receiver-Based Reliability.

4.4 CRIMP Design

Using the methods of reliable many-to-many delivery discussed above, we designed a reliable multicast layer which conforms to the requirements of the architecture. We also use the simplifications that the architecture afforded us to further enhance the design. In this section, we describe the design decisions of CRIMP, our multicast library.

4.4.1 Design Overview

Since we required low latency recovery, we used the Receiver-based reliable multicast as described above. Our implementation borrows heavily from SRM [4]. The major criticism [11] of this approach is the scalability of state kept by the senders, the requirement of infinite storage for packets, and session messages to avoid missing ends of packet sequences. Since the senders in our architecture are limited in number, keeping state for these is not unreasonable. Since packets have a limited lifetime, infinite storage is also not a problem. Packets are part of an unending stream, and thus session messages are unnecessary. Since the simplifications afforded by the architecture cancel these issues, we felt that a Receiver-based approach with slight augmentations could provide us with a scalable, lower latency solution. In the following subsections, we discuss the details of the multicast layer, and the augmentations we made to provide potentially lower latency.

4.4.2 Bootstrapping

In CRIMP, since we limit the number of servers and each server needs a unique identity, we require that new servers establish themselves before sending game messages to the multicast channel. The unique identity is required since packets are referenced, akin to SRM, by the tuple {server identifier, sequence number}. Using the überserver abstraction, we defined a bootstrapping sequence for establishing a mirror in the group.

1. When a mirror wishes to join the game, it sends a request-join control packet to the multicast channel.

2. The überserver responds back to the mirror, on the multicast channel with either an establishment message or a denial message. The establishment message contains the mirror's identifier.

3. Upon establishment, the mirror requests the game-state from the überserver.

4. The überserver replies via unicast TCP with the game-state.

If the messages in steps 1 or 2 are lost, the mirror will retransmit its request based on an exponentially

backing off RTO. These control messages are done on the multicast channel so other mirrors can add the new mirror to their server tables. The überserver will reply with the game-state instead of another mirror, since the game state at the überserver is considered to be the reference state. This is done via TCP to simplify the reliable transmission of a large chunk of data. The mirror will capture packets on the multicast channel while the transfer of game-state is occurring, and apply the changes once it finishes. Once the changes are applied, the mirror can become an active participant in the game. [1]

4.4.3 Loss Detection

Since packets are sent by each mirror at a relatively high rate, at least one packet per 30 ms, detecting loss can be done quite quickly by counting packets received after the next expected sequence number. This scheme, which we borrow from TCP's duplicate ACKs, we call N*-Out-Of-Order*, where N corresponds to the number of packets received after the expected one. If N is chose to be to low, packets will be falsely assumed to be lost. If N is chosen to be too high, packets will be discovered lost too late.

4.4.4 Recovering from Loss

The loss recovery algorithm which we use is very similar to that of SRM. Like SRM, when a lost packet is detected, the receiver schedules a recovery message to the multicast channel on the interval $[C_1 d_{S,A}, (C_1 + C_2) d_{S,A}]$. As in [4], $d_{S,A}$ is the approximate one-way delay from the source server S to the receiving server A as estimated by A. C_1 and C_2 are parameters to the equation which tune how many duplicate recovery messages are sent versus the time waited. In our case, we wish to set C_1 and C_2 aggressively, to 1.5 and 1, somewhat more aggressively than in SRM to avoid latency. As in SRM, if the receiver receives either the packet or a recovery request for the same packet from a different server, nothing is done. Otherwise, the receiver sends a recover request for the packet to the multicast channel. Our enhancement of this recovery algorithm is to have the receiver send an immediate NAK with probability $\frac{1}{n}$ where n is the number of mirrors. By doing this, we expect to always have an immediate NAK if no mirrors receiver the packets. This should add a negligible amount of traffic, in that in the worst case, only 1 extra packet is expected.

To repair the lost packet, any server which receives the recovery request and has the packet stored becomes a candidate for repairing the lost packet. As in SRM, the candidate will schedule a repair packet randomly on the uniform distribution $[D_1 d_{A,C}, (D_1 + D_2) d_{A,C}]$. $d_{A,C}$ is the approximate one-way delay from the server A which sent the recovery request to the repair candidate C as estimated by C. D_1 and D_2 are parameters to the equation which tune how many duplicate repairs are sent versus the delay in receiving a repair. Ideally, these parameters provide a single low latency repair packet. We choose $D_1 = 2$ and $D_2 = 1$ as they for SRM. If a repair is seen for the packet before the time expires, no repair is sent. Otherwise, a repair is sent when the timer elapses. We enhance this further by having the sender respond to recovery requests immediately. This will provide a low latency solution to all receivers, or the set of receivers close to the sender missing the packet. This adds only the cost of a packet per recovery request to the multicast channel.

To avoid responding to recovery requests for the same lost packet, every candidate server which sees a

[1]Since the focus of the paper is on the asymptotic performance and not startup cost, for most of our studies, implemented a simplified the überserver-mirror transaction, by having the mirror block until the überserver register all games. This allowed for a more deterministic study of both the network layer and the overall architecture

repair message will ignore recovery requests for that packet for $Ed_{S,C}$. E is a parameter which tunes how many duplicate repairs are sent for the same recovery request versus how frequently a fresh recovery request will be ignored. We chose $E = 3$, as in SRM. We think that this gives a good balance of ignoring duplicate messages and replying to valid messages.

A server which either sends a recovery request or witnesses a recovery request from a different server must wait for a repair to return. It is possible to lose either the recovery message or the repair, so a retransmission timeout is set for the waiting server. This RTO is $Fd_{S,A}$ and is started when the recovery request was sent or witnessed. Ideally, a repair should return in about an RTT to the sender plus the wait-time needed to avoid duplicate repairs. If this timer expires, the server restarts the recovery process. We chose $F = 2$, corresponding to about an RTT.

4.4.5 Canceling Recovery

Since packets are only useful for a finite slice of time, a server can stop trying to recover a lost packet after a certain amount of time. This is facilitated through the clocking built into CRIMP. Packets and first recovery tries are stored in a data structure called a CRIMP interval. CRIMP intervals act as a history of what has occurred on the multicast channel. When a new packet arrives, it is copied into the newest CRIMP interval. Periodically, the server will call a function `tic()` which moves the newest CRIMP interval into the past and provides a newer CRIMP interval. The oldest CRIMP interval will be removed, and any stored packets in the interval are deleted and recovery requests are canceled. The length of the history is configurable. In our case, it makes sense to set the length of the history of intervals to that of the longest synchronizer, since packets are of no use after that point.

4.4.6 Server Management

CRIMP also handles the problems of network partitioning and server crashes. This is done using the authority of the überserver. The überserver maintains a list of last contacts with servers. If packets are not received in a time corresponding to $\frac{3}{4}$ of the length of time that its CRIMP intervals correspond to, the server is considered dropped. This time is used, since after that point, the server would have difficulty recovering from losses if it was down. Hence, any servers which can reach the überserver are still considered part of the game. If the network partitions, all servers in a section other than the überserver are dropped. If the partitioned servers return after being dropped, they must go through the bootstrapping process again to carry the game.[2].

4.5 Server Selection

Since each game will have a different set of servers for it, we needed a system for a user to be directed to a game mirror. We do this with a master server which tracks games and a graphical tool `qm-find` which interfaces with the master server.

4.5.1 Master Server

When a new game is started by an überserver, it registers with a well-known server, called a master server (a term borrowed from [2]). The überserver sends the ip address and port for itself and its mirrors, as well as the game information being played on the server (map, game style, etc.). The master server sends periodic keep-alive messages via TCP to the überserver which replies with any changes in its relevant information such as new mirrors, removed mirrors, players, or maps. If the überserver does not

[2]We do not consider network partitioning in our evaluation, the description is here only for completeness

reply to a keep-alive message, the master server removes it from its database. The master server also has a client interface which will return games and their lists of mirrors.

4.5.2 `qm-find`

`qm-find` is our proposed tool for finding the closest game to a client. The tool will query the master server for all games matching a specific query (map, game style, etc.). The master server returns the corresponding list of games. When the user selects a game, `qm-find` queries an IDMaps [6] HOPS server to find the mirror with the lowest approximate RTT relative to the user. `qm-find` then spawns the game client with the the lowest RTT mirror as the server name and port arguments.

4.6 Evaluation

Since CRIMP was designed with the architecture in mind, we wanted to evaluate the network layer under game conditions. However, due to limited equipment, and test subjects, we were unable to run experiments with large numbers of clients and servers. Nevertheless, we captured the traffic on the multicast channel for a smaller number of clients, and created a traffic model based on the patterns in these.

In figure 4.1, we present the network traffic for two servers on a 100baseT LAN. Here our definition of traffic is bytes sent per packet. We considered multiple games with varying numbers of clients. Observe that the packet size after some initial large messages, asymptotically remains nearly constant. We also present this data as a histogram in Figure 4.2. We removed the outliers (over 100 bytes) from the plot since we care about asymptotic performance. We used the remaining data points in the histogram as a traffic model for studying multicast performance.

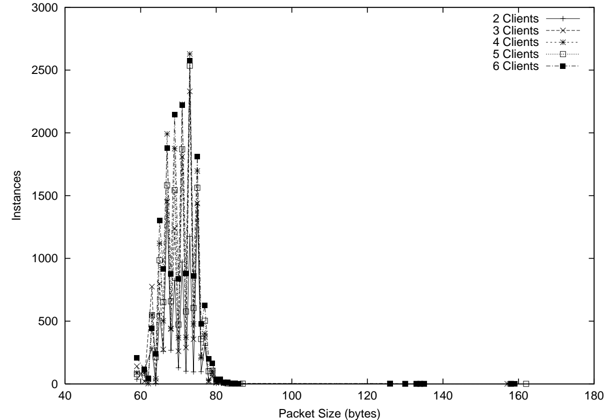


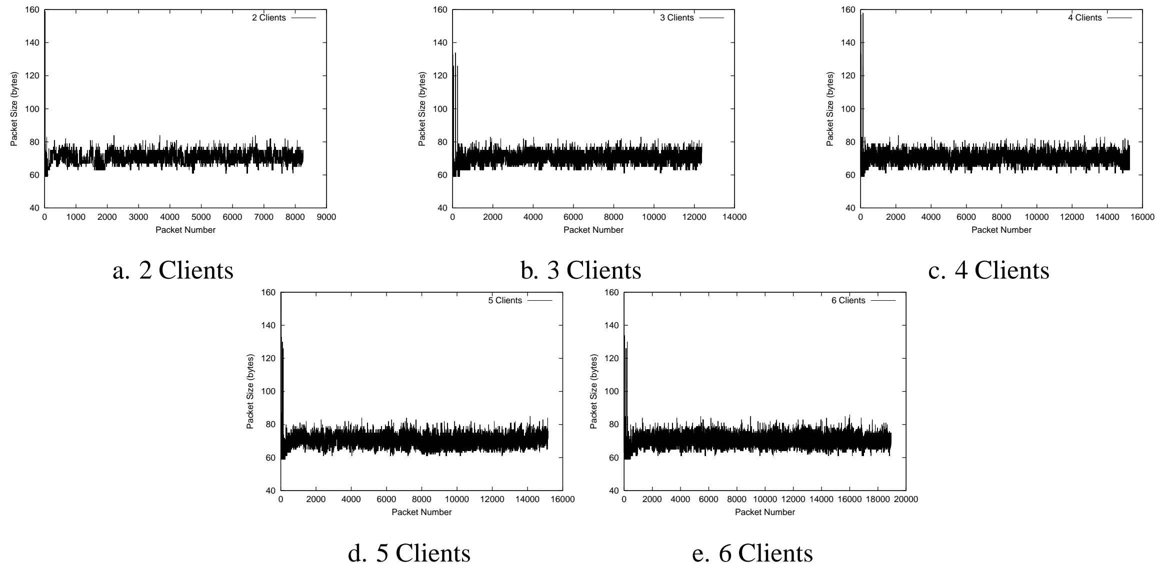
Figure 4.2: Histogram of Packet Sizes

Experimental Setup

Using the traffic model as discussed above, we created a simple test harness to evaluate the network layer. This test harness sent a packet of a size determined randomly, using the histogram as a probability density function, to the multicast channel. Due to limitations of dummynet, we were unable to create random network delay and loss on a multicast channel between two machines. We ran multiple test harnesses on the machines to create a virtual topology like that in Figure 4.3.

Using this virtual topology, having a two-way FreeBSD Dummynet [17] bridge (denoted Dummynet in the figure) with a delay of 25 ms, and variety packet loss rates, we evaluated the multicast layer in terms of perceived RTT, total losses and duplicate ACKs. Each sender sent 2000 total packets, one packet every 30 ms, as in Quake. We initialized CRIMP with a history of 1000 ms.

To compute meaningful results, we recorded the send time of each packet, and the time that each client receives the packet. We also recorded the number of duplicate recovery packets sent and number



a. 2 Clients b. 3 Clients c. 4 Clients

d. 5 Clients e. 6 Clients

Figure 4.1: Two-mirror Client Traces

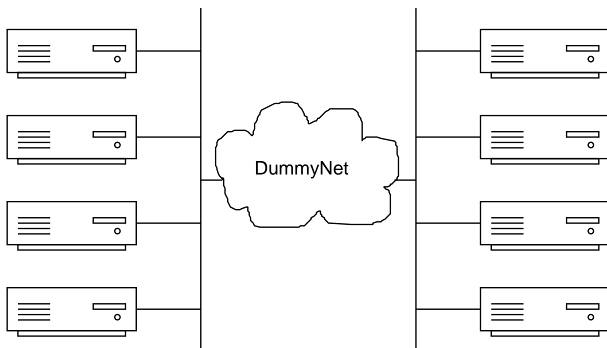


Figure 4.3: Virtual Multicast Topology

Loss Rate	Perceived RTT	Total Losses	Duplicate Recovery	Duplicate Repairs
0%	15ms	0	0	0
5%	32ms	0	441	323
10%	78ms	0	1653	1215
15%	161ms	2	2512	1951

Table 4.1: Performance of Multicast

of duplicate repairs. We consider a packet to be a duplicate if a packet with the same data has already been sent. Hence, if a receiver sends two recovery requests, one is considered to be a duplicate. We present the results from our experiment in in Table 4.6. We concede that the topology may cause the multicast layer to exhibit unusual results since half of the machines in essence see the same behavior (25ms delay, a loss, or 0ms delay). This nullifies any chance of localized recover, and also causes the probability of no one receiving the packet 0, in essence making our aggressive recovery less effective.

From these results, we can see that our multicast adequately provides reliability for our architecture. Note that only in the 15% loss rate case were packets not received. We think that this may be misrepresen-

tative, and an artifact of our topology having an entire side miss a packet. In the 5% loss case, which we view as a conservative estimate of an actual network, we experience a perceived RTT about than twice that of the 0% case. This behavior is consistent with our expectations of an efficient reliable multicast.

4.7 Future Work

4.7.1 Forward Error Correction

A possible improvement for recovery is using forward error correction [10] to recover packets without going to the network. If, for example, each packet also included the contents of the last two packets, the server would in essence, have received all three packets. If one of the previous two packets was lost, it could be recovered from that packet. The network-based reliability would thus be needed only for bursts of packet loss. This would decrease recovery latency for some packets at the expense of increased network stress (in the example, packets would triple in size).

Chapter 5

Conclusions

In conclusion, we present a novel networked game architecture, the Mirrored-Server architecture. This architecture takes advantage of mirroring that has been shown to provide better services to clients [6], and extends it to a system which changes at a very high rate. To facilitate this architecture, we also describe a new synchronization mechanism, Trailing State Synchronization to provide efficient consistency between mirrors. We also present a reliable multicast layer, CRIMP, which takes advantages of simplifications presented by this architecture to provide low latency many-to-many reliability. We also some results which prove that our architecture can provide quality gameplay, yet leave much evaluation for future work, since we were hamstrung by the oddities of the Quake server.

Bibliography

[1] N. Baughman and B. Levine. “cheat-proof playout for centralized and distributed online game s”. In *Proc. Infocom 2001*, April 2001.

[2] Y. W. Bernier. “Half-Life and Team Fortress Networking: Closing the Loop on Scalable Network Gaming Backend Services”. In *Proc. of the Game Developers Conference, 2000*, May 2000.

[3] Diablo II, Starcraft. Blizzard entertainment, January 2001. `http://www.blizzard.com/`.

[4] S. Floyd et al. “A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing”. In *IEEE/ACM Transactions on Networking*, Dec. 1997.

[5] EverQuest, PlanetSide. Verant interactive, January 2001. `http://www.verant.com/`.

[6] P. Francis, S. Jamin, C. Jin, Y. Jin, V. Paxson, D. Raz, Y. Shavitt, and L. Zhang. “IDMaps: A Global Internet Host Distance Estimation Service”. Submitted for publication, 2001.

[7] T.A. Funkhouser. “ring: A client-server system for multiuser virtual environments”. In *Proc. of the SIGGRAPH Symposium on Interactive 3D Graphics*, pages 85–92. ACM SIGGRAPH, April 1995.

[8] L. Gautier and C. Diot. “Distributed Synchronization for Multiplayer Interactive Applications on the Internet”. Submitted for publication, October 1998.

[9] D. Helder and S. Jamin. “Banana Tree Protocol, and End-host Multicast Protocol”. Submitted for publication, July 2000.

[10] J.F. Kurose and K.W. Ross. *“Computer Networking: A Top-Down Approach Featuring the Internet”*. Addison Wesley, Preliminary edition, 2000.

[11] B. N. Levine and J. J. Garcia-Luna-Aceves. “a comparison of reliable multicast protocols”.

[12] D. Li and D. Cheriton. “OTERS (On-Tree Efficient Recovery using Subcasting): A reliable multicast protocol”. In *Proc. IEEE International Conference on Network Protocols (ICNP’98)*, pages 237–245, 1998.

[13] J. Liebeherr and T. Beam. “hypercast: A protocol for maintaining multicast group members i n a logical hypercube topology”. In *Proc. of the 1th Workshop on Networked Group Communication*, pages 72–89, 1999.

[14] Katherine L. Morse. “interest management in large-scale distributed simulations”. Technical Report ICS-TR-96-27, 1996.

[15] L. Pantel and L. Wolf. "On the Impact of Delay on Real-Time Multiplayer Games". Submitted for publication, 2001.

[16] Quake. id software, January 2001. `http://www.idsoftware.com/`.

[17] L. Rizzo. "Dummynet: A Simple Approach to the Evaluation of Network Protocols". *ACM Computer Communication Review*, 1997.

[18] J. C.-H. Lin S. Paul, K. K. Sabnani and S. Bhattacharyya. "Reliable Multicast Transport Protocol (RMTP)". *IEEE Journal of Selected Areas in Communications*, 15(3):407–421, 1997.

[19] B. Stabell and K. R. Schouten. "The Story of XPilot". ACM Crossroads, January 2001.

[20] S. G. Rao Y. Chu and H. Zhang. "A Case for End System Multicast". In *Measurement and Modeling of Computer Systems*, pages 1–12, 2000.