Low Latency and Cheat-proof Event Ordering for Peer-to-Peer Games*

Chris GauthierDickey,† Daniel Zappala, Virginia Lo, and James Marr‡
University of Oregon
Department of Computer Science
1202 University of Oregon
Eugene, OR 97403-1202
{chrisg | zappala | lo | james}@cs.uoregon.edu

ABSTRACT

We are developing a distributed architecture for massively-multiplayer games. In this paper, we focus on designing a low-latency event ordering protocol, called NEO, for this architecture. Previous event ordering protocols prevent several types of cheats at the expense of operating at the latency of the slowest player. We broaden the definition of cheating to include four common protocol level cheats and demonstrate how NEO prevents these cheats. At the same time, NEO has a playout latency independent of network conditions and adapts to network congestion to optimize performance.

Categories and Subject Descriptors: C.2.4 [Distributed Systems]:Distributed applications, I.6.8 [Types of Simulation]:Gaming

General Terms: Algorithms, Performance, Security

Keywords: low-latency, cheat-proof, peer-to-peer, distributed, interactive, games

1. INTRODUCTION

Traditionally, multi-player games have used a client/server communication architecture. This architecture has the advantage that a single authority orders events, resolves conflicts in the simulation, acts as a central repository for data, and is easy to secure. On the other hand, this architecture has several disadvantages. First, it introduces delay because messages between players are always forwarded through the server. Second, traffic at the server increases with the number of players, creating localized congestion. Third, with small multi-player games, the server is hosted by one player, and the others must trust that the server is not tainted. Last, this architecture is limited by the computational power of the server. While we can throw technology at most of these problems in the form of more servers and higher bandwidth lines, this solution incurs significant cost.

To address these problems, we are developing a fully distributed, peer-to-peer architecture for massively-multiplayer online games (MMOGs). This architecture allows peers to send messages directly to each other, reducing the delay for messages and eliminating localized congestion. It allows players to start their own games without the incredible investment in resources required by a client/server architecture. Furthermore, this architecture allows games to overcome the bottleneck of sever-only computation by harnessing the processing power of the players' machines through peer-to-peer computing.

To build a distributed game, we must first overcome the fundamental problem of preventing cheating in an untrusted environment. Specifically, how can players trust each other to accurately represent when a given event has occurred? Accordingly, the first component we have designed for this architecture is the New-Event Ordering (NEO) protocol, which provides low latency event ordering while still preventing common protocol-level cheats. NEO provides much lower latency than previous event ordering protocols, which are limited by the latency of the slowest player to any other player in the game. NEO divides time into "rounds" and uses the round duration to bound the maximum latency of a player *from a majority of other players in the game.* This means that it is acceptable to be slow to some players, as long as most players get your updates in a timely fashion. While NEO dramatically improves performance, it does not compromise trust. We show how NEO can prevent five common protocol-level cheats, under a broader definition of cheating than has been previously used.

In this paper, we describe our peer-to-peer game architecture and the motivation behind designing a new architecture. We then describe the NEO protocol and show how it can provide low latency event ordering for distributed games, while also preventing cheating at the protocol level. We finish by extending NEO with several enhancements that allow players to react to network congestion and improve game play when congestion is low.

*This work was supported in part by the National Science Foundation under grant ANI-9977524.

†Supported by an NSF Graduate Research Fellowship

‡Supported by an NSF Research Experience for Undergraduates grant

NOSSDAV'04, June 18–18, 2004, Cork, Ireland.

2. A TAXONOMY OF CHEATING

In order to understand the problems that arise when designing a distributed event ordering protocol, we present a short taxonomy of cheating. We define a cheat as any action by a player that gives her an unfair advantage over another player. Cheats can be categorized by the layer in which they occur: game, application, protocol, or network. The cheats are described in the context of the cheating player named Eve and a competing player named Alice.

Game level cheats occur by breaking the rules of the game. For example, Eve discovers that by dropping an object while casting a spell allows her to keep a copy of the object in her inventory, even though she just dropped it (granting her the ability to duplicate any object in the game). Application level cheats occur by modifying the code of the game or operating system. A common example is modifying the rendering code so that walls in a game are invisible, making it easy to locate hidden players. Protocol level cheats occur by modifying the protocol, such as changing the contents of packets. Network level cheats occur because of properties inherent in the network layer. A denial-of-service attack is an example of a network level cheat.

Our focus in this paper is preventing protocol level cheats. We define five common protocol level cheats, based on our experience with distributed games:

Fixed-Delay Cheat: In the fixed-delay cheat, Eve adds a fixed amount of delay to her outgoing packets, allowing Eve to receive packets faster than she is sending them. Eve gains the advantage of being able to react quicker to other players than they can react to her delayed packets.

Timestamp Cheat: Because events must be ordered for consistency purposes, a global clock is often used for time stamping. In the timestamp cheat, Eve waits to receive an update from Alice and then sends her update with a timestamp that is *before* Alice's. For example, Eve could send out a move with a timestamp earlier than the 'Alice shoots Eve' update just received. To other players, Eve's message appears to be delayed and the shot misses.

Suppressed Update Cheat: In this cheat, Eve suppresses all updates to one or more players, while continuing to receive their updates. This cheat allows Eve to 'hide' from other players, since they are no longer receiving her updates, but she is receiving theirs. Eve sends a new packet with her current position before she is dropped from the game.

Inconsistency Cheat: In the inconsistency cheat, Eve sends different updates to different players in the game. We describe this cheat in the context of Alice, a competing player. Eve sends her 'real' update to every player while sending a different update to Alice at time t. Now Alice thinks Eve is in a different location than she really is, but every other player will disagree with Alice on Eve's location. Later, Eve can send updates to Alice that merge the two differing opinions on her location in order to hide her cheat. In the worse case scenario, Alice can corrupt an entire game, but Alice can also corrupt a single player, eliminating them from the game. The inconsistency cheat arises from the Byzantine General's Agreement problem [10], but in this case we are trying to have an agreement on everyone's game state.

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

3. MOTIVATION

Besides overcoming the limitations of the client/server architecture, we are motivated to develop a fully distributed architecture for MMOGs because of the past research related to distributed games.

Diot, Gautier and Kurose described the first protocol for distributed games in [6, 8] and built a game called MiMaze to demonstrate its feasibility. Their work is important because they developed a technique called bucket synchronization, in which game time is divided into 'buckets', in order to maintain state consistency among players. The MiMaze protocol uses multicast to exchange packets between players, resulting in a low latency; however, it does not address the problem of cheating.

At the other end of the spectrum, Baughman and Levine designed the *lockstep* protocol to address the problem of protocol level cheats [1]. Lockstep uses rounds for time, which are broken into two steps: first, everyone reliably sends a cryptographic hash of their move, then everyone sends the plain-text version of their move. This forces everyone to *commit* their move, without revealing it, thereby preventing anyone from knowing someone else's move ahead of time.

To mitigate the problem of delay introduced by reliable transport, Baughman and Levine added event scoping (called asynchronous synchronization in [1]). Event scoping requires that players exchange updates only when their actions might intersect. To do this with lockstep, each player associates a sphere of influence with every other player. When a player receives or misses an update from another player during a round, the associated sphere is contracted or dilated respectively. This allows players to progress in rounds asynchronously until their sphere intersects with another player's sphere–at which point they must engage in lockstep (and wait for each other's messages).

Lockstep is a major advance in distributed protocols because it is provably secure against the fixed-delay and timestamp cheats. It gains this security by forcing moves to occur in lockstep–no player can receive a plain-text move before they commit their move.

Unfortunately, lockstep has several drawbacks. First, its *playout latency*, which is the time from when an update is sent out to when the update can be displayed to other players, has a minimum bound of three times the latency of the slowest link between any two players. This bound is due to the use of reliable transport for sending the hashed update, followed by sending the plain-text update. Event scoping does not help to reduce latency because players that are close in the virtual world of a game may in fact incur significant propagation and queueing delay. Second, lockstep is vulnerable to the suppressed update cheat–a malicious player can stop sending updates, stopping round progression until other players drop her from the game. Last, lockstep is vulnerable to collusion when event scoping is used. Since rounds no longer progress synchronously, a player can receive a plain-text update from another player and forward the update on to other players who have not yet committed their move for that round.

Cronin et al. designed the Sliding Pipeline (SP) protocol [4] in order to improve the lockstep protocol. They add an adaptive pipeline that allows players to send out several moves in advance without waiting for ACKs from the other players, reducing the time that is dead-reckoned between rounds. The pipeline depth is designed to grow with the maximum latency between players so that *jitter*, or inter-packet arrival time, is reduced.

While the SP protocol reduces jitter and dead-reckoning, it still has the same playout latency as lockstep. In terms of security, the protocol prevents the timestamp cheat, but allows a player to use the fixed-delay cheat [4] because a player can artificially increases her delay to receive a plain-text move before committing her move for a given round. The adaptive pipeline helps detect this cheat, but it can falsely label someone with an increased delay as a cheater. Furthermore, a cheater can use the fixed-delay update cheat every other round and not be detected.

Bharambe et al. have proposed Mercury, a distributed publish-subscribe communication architecture [3]. Mercury provides channels, which can be of any subject, uses a subscription language (that is a subset of relational database query languages), and uses rendezvous points (RPs) to gather and disseminate publications. Unfortunately their results show that it cannot meet the performance requirements for MMOGs due to the routing delay introduced by their architecture [3].

In the game industry, very few networked games are fully distributed. One notable exception is Age of Empires (AoE) [2], in which games are synchronized across clients and peer-to-peer communication is used. AoE's protocol is similar to bucket synchronization, except that unicast is used. While AoE is a commercial success for distributed game protocols, it is subject to all but the inconsistency cheat (because players periodically exchange hashes of the game state with other players to detect inconsistencies).

Finally, we note that the area of distributed interactive simulation (DIS) addresses some of the same issues, but all participants are trusted, so the DIS protocols do not attempt to prevent packet-level cheating.

4. TOWARDS A PEER-TO-PEER GAME ARCHITECTURE

We are developing a fully distributed MMOG architecture based on four components. First, an authenticating component is responsible for allowing or denying players access to the game. Second, a communication component determines how players send messages to each other. Third, the storage component provides long-term storage for world state. Last, the computation component determines how computations are distributed to players in the game. Figure 1 shows our architecture.

The authenticating component provides several necessary functions. First, every player is uniquely identified in the game, through a mechanism such as a public-key system or registration in the game with something such as a credit card number. This feature allows the system to determine if any two identifiers belong to the same individual. Further, it allows the game to permanently remove players and prevents a single player from joining the game simultaneously with multiple identities. Second, we assume the authenticating component can generate keys for the game and authenticate

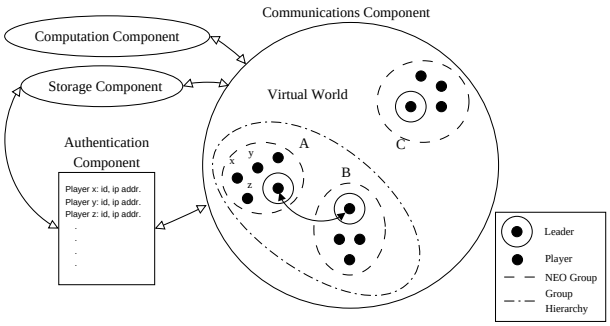

Figure 1: Our peer-to-peer game architecture

players. We call this function the *authenticating directory* (AD) service, which allows an authenticated player to locate current players and allows authenticated players to present proof that a player was cheating in order to permanently ban another player from the game.

The communication component uses a self-organizing hierarchy that mirrors the structure of the virtual world. This model works well with games because events in one part of the world do not affect other parts of the world. In each part of the virtual world, players form a peer-to-peer network in order to exchange event messages. These networks are at a very small granularity, such as a room within a building, as determined by how far events must propagate to reach the affected players. Some events affect multiple groups, such as the lights going off in a building; in these cases, the event is propagated through the world hierarchy by group leaders in each peer-to-peer network.

The storage component, which stores long-term game state, is comprised of a distributed hash table (DHT), such as CAN [13] or Chord [14], and security and reliability services. A large amount of research is currently being performed on peer-to-peer file systems (such as OceanStore [9] or CFS [5]), and we hope to leverage this work and determine how it meets the needs of a distributed MMOG. Particular concerns include long-term persistent storage and fast and secure updates.

The last component, the computation component, is used to schedule game computations among the players. We are currently researching scheduling techniques [11] to determine efficient algorithms to schedule computations on players' processors, though games have unique requirements. For example, the artificial intelligence (AI) of a monster should be activated when a player approaches it within some distance. However, the players directly interacting with the monster cannot be trusted to accurately compute the AI of the monster. On the other hand, the AI should be verified by several other players to make sure that collusion doesn't occur between distant players. The implication of these needs is that a scheduler must be able to make decisions about who can execute processes in the game and results must be verified to prevent cheating.

5. LOW DELAY EVENT ORDERING

Our current focus is on designing the communications component for our distributed game architecture. In particular, we are focusing on event ordering at the lowest level in the communications hierarchy–within a group of players

that may be in the same room of a building. To provide this local ordering, we use a global clock to mark when events occur, but this leads to the challenging problem of trusting whether an event actually occurred at the time a player asserts that it has. In the rest of this paper, we focus on this problem in isolation; we recognize that several challenging problems remain, such as splitting and merging groups of players, and propagating events along the communications hierarchy.

Lockstep is the first event-ordering protocol to address the issue of cheat prevention. To guarantee that events have occurred at the stated time, lockstep orders events by rounds, incrementing a round only after every player has committed their move for that round. The price of this total ordering of events is a delay that is proportional to the largest delay between any two players.

To understand this problem more clearly, let us assume we have a multi-player game with a group of players located in the US and all in the same virtual location in the game. Under the lockstep protocol, the players must all exchange messages. At time t, a new player arrives in the same virtual location, but is connecting from Mongolia. A quick measurement shows the average round-trip time from the US to Mongolia is 728ms. This new player will force *all* players to proceed in lockstep, which requires 3 times the longest delay between players (due to the use of reliable transport). Assuming links are symmetric, rounds will progress at the rate of 1 round per 1,092ms, assuming that *no* packets are lost. Typically, the desired round time is an order of magnitude smaller [6].

Our protocol, inspired by *bucket synchronization* in [6], uses rounds to order events. The length of a round is bounded by a maximum latency, ensuring that players can receive updates in a timely fashion. To prove that an event occurred at its stated time, a player must be able to send her update to a majority of other players within the round duration. This proof is communicated to all the players through a round of voting, indicating which messages each user has received during the round duration. As long as a majority of players receive updates on time, then with the situation above, the slow player from Mongolia will not affect the round duration. The tradeoff is that a player who is slow to most other players will also not be able to play in this area of the virtual world. However, we feel this tradeoff is preferable to making the game unplayable for everyone.

6. NEW-EVENT ORDERING PROTOCOL

The New-Event Ordering (NEO) protocol is the first protocol that totally orders events generated by a distributed group, avoids five common protocol level cheats, has a playout latency that is independent of network conditions, and adapts to changing network conditions to optimize its performance.

NEO is purposely agnostic regarding the underlying message propagation system. Unicast, multicast, or some type of overlay could be used to send messages, though the use of something other than unicast or native multicast could introduce new ways of cheating (such as not forwarding messages). However, this flexibility allows us to have alternatives when considering the extreme case of everyone in a game going to the same location in the virtual world and having to exchange messages. In this case, group density could be used to trigger a switch to multicast, for example.

With NEO, the majority always rules. This has the benefit that the protocol will adapt so that the majority of players are receiving the best possible performance. However, this also means that if a majority of players can collude and cheat, then NEO will not be able to prevent it. We address this problem under *Collusion Cheats* below.

In our discussion of NEO, we assume that all players are in the same location of a virtual world, that all players know of each other and communicate via UDP over unicast, that any player can authenticate the message of another player through signatures, and that game time is synchronized between players using a time synchronizing protocol such as NTP [12].

6.1 Basic NEO Protocol

For simplicity, we start with a basic NEO protocol that prevents only the suppressed update and timestamp cheats. We later extend this protocol to address the other three protocol level cheats.

In NEO, time is broken into equal intervals, called rounds, in which each player sends an update to all other players. Each update is encrypted, and in the following round, each player sends the key for the previous update to all other players.

NEO uses rounds in order to bound the maximum delay that any player can have for sending their update. Late updates are considered invalid, unless a majority of other people have received them. This means that unlike the lockstep or sliding pipeline protocol, which have playout latencies bounded by 3 times the maximum latency between any two players, NEO bounds its playout latency by only $2d$, where d is the round length and is *independent of any player's latency.* This allows game developers to choose how big or small the round length is, and therefore the responsiveness of the game.

Presumably, the maximum round length is the maximum amount of time a round can be for a game to be playable. Thus, any player who cannot reach a majority of players within the maximum round length cannot play the game with those players. This is acceptable, since the game is unplayable when a player's delay is beyond the maximum round length.

Each message contains a time-stamped, signed, encrypted update, a key for the previous round, and a signed bit-vector of messages received from the previous round. For example, a message M from player A at round r has the following format:

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

In this message, $E(x)$ is an encrypted x, $S_A(x)$ represents A's signature on x, U_A^r is the update from player A for round r, K_A^{r-1} is A's key for the update from round $r-1$, and V^{r-1} is the bit vector of votes for messages (defined shortly) received during round $r-1$.

Because a player releases her key for an update immediately after the end of the round, she cannot accept any late updates. However, each player may have a different set of updates that arrived on time for a given round. To maintain consistency, players accept an update only if a *majority* of players received the update on time.

Consistency is achieved through a distributed voting mechanism. A player votes positive for another player if the other player's update was received on time; otherwise, she votes negative. An update is considered valid only if a majority

Table 1: Player A's Table of Votes

Player	Bit-vector
A	1 1 0 1 0
B	0 1 0 1 0
C	1 1 1 0 0
D	1 1 1 1 1
E	*packet lost*
Voting tally	3 4 2 3 1

of the players send a positive vote. Each round the players tally the votes they have and decide which updates are considered valid. Any votes which are not received are considered abstentions; however, a majority of votes must be received for the vote to be considered valid. If not enough votes are received, the players must attempt to contact the players that abstained from the vote.

To understand how voting works, assume five players are in a game, and player A is tallying the votes from the previous round. Also assume that a majority is greater than 50%. Table 1 lists the voting bit-vectors that each player has sent to player A. From the tally, we can conclude that a majority received A, B and D's updates, while a majority did not receive E's update (so it is considered invalid). As for player C, player A cannot determine what the outcome of the vote is, so she must contact another player to determine the outcome.

The primary reason for voting is that it allows rounds to progress without needing to hear from every player every round. This decouples the playout latency from the players' latency because round progression no longer relies on reliable communication. Assuming that a majority of players are receiving updates and votes, NEO will continue to progress through rounds[1]. On the other hand, with lockstep and the sliding pipeline protocols, if just *one* player drops an update, all players must wait until that update is recovered before the game can progress to the next round.

The secondary reason for using voting is that we only want to reconcile a minority of players at any time in order to keep the majority of players happy. Recall that dead-reckoning is being used between rounds so that if a player has to adjust their simulation, it is because she is with a minority of players whose game state differs from the majority.

In [7], we prove the safety and liveness of NEO, but omit the proof here due to space constraints. The safety and liveness proof tells us that the fixed-delay and timestamp cheats are not possible under NEO and that NEO always progresses. Safety can be understood intuitively because a key is never sent until the round is over, at which point no new moves for the round can be generated. We prove liveness by showing that round numbers increase monotonically with real time and that NEO does not halt for any reason, even in the face of inadequate votes.

6.2 NEO with Pipelined Rounds

In the basic protocol, the delay from each player to the majority of other players is bounded by the duration of the round. Increasing the round length increases the frequency with which the game must dead-reckon the positions and

[1]If a majority of players are not receiving updates from each other, then the game is unplayable. But this holds true for any game, distributed or not!

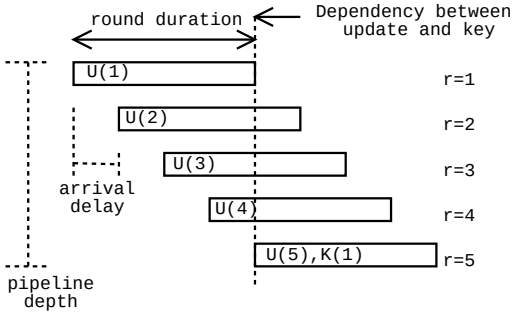

Figure 2: Pipelining rounds in NEO.

actions of other players. During this period of dead-reckoned time, the game is inconsistent and unresponsive. To address these problems, NEO pipelines its rounds, similar to the technique of pipelining instructions in a processor and to the SP protocol [4]. The pipeline depth is related to the round duration and the round arrival delay, as seen in Figure 2. This relationship can be expressed in the following formula:

$$pipeline\ depth = \frac{round\ duration}{arrival\ delay} \tag{2}$$

Using pipelined rounds does not significantly change our basic protocol, except with regard to sending out the key to our encrypted update and how often updates are sent out. A dependency exists between the end of the round that an encrypted update is sent out and the beginning of the round that the key is sent out (see Figure 2). Similar to a dependency in a processor pipeline where we must wait until the dependency has passed to execute a new instruction, we must wait until the round with the update has passed before we can send the key for the update. For example, if a round starts at t=80ms and the round duration is 120ms, then the key must not be sent until t=200ms. We can now generalize Equation 1 using the pipeline depth d and round number r for player A in the following equation:

$$M_A^r = E(S_A(U_A^r)), K_A^{r-d}, S_A(V_A^{r-d}) \tag{3}$$

As the sending rate of updates increases, the responsiveness and visual smoothness of the game increase.

6.3 Security

Now we explain how NEO prevents the cheats from Section 2:

Fixed-Delay Cheat: NEO addresses this cheat through the use of bounded round lengths. Late updates are simply ignored by everyone.

Timestamp cheat: NEO prevents this cheat through the use of bounded round lengths. Once a round has passed, a player can no longer submit a move for that round; therefore it is impossible to receive a decrypted update before submitting the move for the previous round.

Suppressed Update Cheat: NEO adjusts the sending rate of Eve's opponents, as described in Section 7. Eve's missing packets signal congestion to NEO, so that her opponents will stop sending their updates to her. Thus, she no longer has an advantage by suppressing updates since she will no longer receive her opponent's updates either. If a player crashes, they will simply be ignored by other players until they are removed from the system.

Inconsistency Cheat: NEO addresses this cheat through the use of digital signatures and state comparison. Players periodically audit game state by performing a state comparison. When two players discover different state, the trail of packets they have received can be used as evidence against a cheating player. Using the authentication component of our architecture, cheating players can be permanently removed from the game once this proof is provided.

Collusion Cheat: NEO addresses collusion at the architectural and protocol levels. First, NEO can adjust the majority value sufficiently high to prevent collusion. Second, the AD service prevents players from logging in multiple times and artificially gaining a majority. Third, the communication component of our architecture can randomly select *witnesses* for a NEO group. As the number of witnesses increase, the probability that a group of colluding players can form a majority decreases.

7. PERFORMANCE ENHANCEMENTS

In order to improve performance and to react to network congestion, we modify NEO to dynamically adjust the round duration and sending rate. To prevent synchronization problems and to re-synchronize disconnected players who have returned, NEO updates include the starting time of the round, the round duration, and the current sending rate. Over the long term, if any player consistently receives late messages, she can re-synchronize her game state with the other players (as when joining the game).

Adjusting the round duration and sending rate is a tradeoff in performance and overhead. Shorter rounds allow games to be more responsive to players, and higher sending rates decrease the dead-reckoned time and jitter. NEO uses peer-to-peer voting to find a consensus for adjustment; more frequent voting produces quicker reaction to network conditions.

7.1 Adjusting the Round Duration

Because players send out their updates at the start of each round, each player can record the delay from other players to herself. Early updates indicate that the round duration can be decreased, from the perspective of that player, while late updates indicate that the round duration should be increased. NEO uses a weighted average over the last several rounds to avoid reacting to transient congestion. Once votes are collected and a majority of votes are for an adjustment, the new round duration and the time for the round change are advertised to all players.

7.2 Adjusting the Sending Rate

In addition to adjusting the round length, NEO should react to congestion as indicated by dropped packets. Every player in the game can measure her own loss rate and other players' late packets. Players can adjust the sending rate locally and globally, to react to short-term and long-term congestion.

A player adjusts her sending rate locally by purposely skipping updates. Skipped updates decrease responsiveness in the game, but due to the voting mechanism in NEO, other players will not need to retrieve her skipped update. Players vote to globally adjust the sending rate in response to long term congestion. Each player keeps a weighted average of their local loss rate. When a majority of votes for a global rate adjustment is collected, the new rate and time of the rate change is advertised to the players.

A player may also adjust her local sending rate in order to prevent the suppressed update cheat. Because a cheater may purposely skip updates, we want to ensure that a player never sends more updates than she is receiving. To achieve this, each person may skip updates to a particular player whenever her rate exceeds that player's rate. Any player that attempts to suppress packets to another player will find that the other player will immediately begin to suppress messages in return.

8. FUTURE WORK

While we can analyze the performance of NEO, certain aspects of NEO are difficult to characterize. We plan to explore NEO's performance through simulation and implementation, comparing NEO's performance with both the lockstep and sliding pipeline protocols. We plan to test NEO's adaptive qualities. Considering the amount of traffic that online games generate as they become more popular each year, we feel that any communication protocol for games should "play fair" with TCP. Beyond NEO, we plan to continue developing our architecture. The next step is to design the protocols for group management and event propagation to test their feasibility. After that, we will design the storage and computation components.

9. REFERENCES

[1] N. E. Baughman and B. N. Levine. Cheat-proof Playout for Centralized and Distributed Online Games. In *IEEE Infocom*, 2001.

[2] P. Bettner and M. Terrano. 1500 archers on a 28.8: Network Programming in the Age of Empires and Beyond. In *Game Developer's Conference*, March 2001.

[3] A. R. Bharambe, S. Rao, and S. Seshan. Mercury: A Scalable Publish-Subscribe System for Internet Games. In *1st workshop on Network and System Support for Games*, pages 3–9. ACM Press, 2002.

[4] E. Cronin, B. Filstrup, and S. Jamin. Cheat-Proofing Dead Reckoned Multiplayer Games. In *Intl. Conf. on Application and Development of Computer Games*, January 2003.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area Cooperative Storage with CFS. In *Proceedings of SOSP*, Oct 2001.

[6] C. Diot and L. Gautier. A Distributed Architecture for Multiplayer Interactive Applications on the Internet. *IEEE Networks*, 13(4), July/August 1999.

[7] C. GauthierDickey, D. Zappala, V. Lo, and J. Marr. Low-Latency and Cheat-Proof Event Ordering for Distributed Games. Technical Report CIS-TR-2004-2, University of Oregon.

[8] L. Gautier, C. Diot, and J. Kurose. End-to-End Transmission Control Mechanisms for Multiparty Interactive Applications on the internet. In *IEEE Infocom*, 1999.

[9] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *ACM ASPLOS*, November 2000.

[10] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. In *ACM Transactions on Programming Languages and Systems*, pages 382–401, July 1982.

[11] V. Lo, D. Zappala, D. Zhou, Y. Liu, and S. Zhao. Cluster Computing on the Fly: P2P Scheduling of Idle Cycles in the Internet. In *IPTPS*, March 2004.

[12] J. Postel. Network Time Protocol. RFC 1305, March 1992.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *ACM SIGCOMM*, 2001.

[14] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM*, pages 149–160. ACM Press, 2001.