Low-Latency and Cheat-proof Event Ordering for Distributed Games

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

***Abstract*— In this paper, we describe a new protocol for ordering events in peer-to-peer games that is provably cheat-proof. We describe how we can optimize this protocol to react to changing delays and congestion in the network. We validate our protocol through simulations and demonstrate its feasibility as a real-time, interactive protocol. To our knowledge, this is the first peer-to-peer protocol that is both cheat-proof and maintains the low latency required by interactive, real-time games.**

Keywords: System design, Simulations

I. 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, in the client/server architecture, players 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 and is not a viable option for the average player who desires to host a game.

To address these problems, we propose using a peer-to-peer architecture [1], [2]. This architecture allows peers to send messages directly to each other, reducing the delay for messages and eliminating localized congestion. It allows a player to host a game without the need for a high-bandwidth, dedicated connection to the Internet. Last, players no longer need to trust an individual server.

Most multiplayer games consist of 2 to 64 players, though a certain class of games, termed *Massively Multiplayer Online Games* (MMOs) scale to several thousand players. In our work, we consider both types of networked games. Our work applies to small networked games and also applies to MMOs as part of a larger communication architecture [2]. In the context of MMOs, our protocol is used for small, virtually local groups that propagate events to other groups as needed.

Building a distributed game communication architecture introduces the fundamental problem of cheating in an untrusted environment. Specifically, how can players trust each other to accurately represent when a given event has occurred? Accordingly, we have designed the New-Event Ordering (NEO) protocol, which provides low latency event ordering while provably 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.

This paper has several important contributions. First, we provide a taxonomy of cheats, explaining where they occur and what can be done to counter them. Second, we contribute NEO–the first cheat-proof, low-latency, congestion controlled protocol for real-time, interactive games. Third, we provide a study of NEO through simulation and demonstrate its superior latency characteristics in comparison to other cheat-proof protocols. Last, we demonstrate NEO's ability to react to changing delay and packet loss in the network.

II. A TAXONOMY OF CHEATING

Before discussing NEO, we present our taxonomy of common cheats in networked, multiplayer games. We categorize the cheats according to the *level* they occur

TABLE I
A TAXONOMY OF CHEATING

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

Check marks indicate whether this type of cheat is possible under the listed architecture. Stars indicate cheats which are partially possible.

at: game, application, protocol, or network. Table I shows this taxonomy and the architectures they occur in.

Cheats at the game level result from manipulating game rules for unintended results. At the application level, the code of the application is modified to give the player an unfair advantage. Cheats at the protocol level occur by modifying network packets or network protocols[1]. Last, cheats at the network level occur because of network security issues–for example, DoS attacks.

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

This taxonomy reveals several interesting things about the way the architecture affects the types of cheating available. First, the client/server architecture by nature prevents a client from sending different packets to other clients, changing the timestamps on packets and learning secrets which are normally not available to the client. However, even though a client/server architecture provides some security for network level cheating, a client can still artificially suppress packets or add fixed delays.

The following sections discuss the categories and their respective cheats in details. In the examples, we have three characters: Alice and Bob, two non-cheating players, and Eve, a cheating player.

[1]Typically protocol and application level cheats both occur through modifying the application, though network cheats specifically target security problems with the network protocols.

A. Protocol level cheats

1) Suppressed Update: In this cheat, Eve suppresses updates to Alice and Bob. Right before the game would disconnect her due to packet loss, Eve sends a new update to her opponents. As a result, Alice and Bob do not know where Eve is exactly or what actions she has performed, giving her the ability to 'hide'.

This cheat is particularly powerful when a player's update includes the actual location of a player instead of the series of moves (which is often the case when using unreliable protocols since a move might have been dropped). In this case, Eve does not let other players know where she is and then picks an advantageous location several rounds later.

2) Fixed Delay: In this cheat, Eve purposely adds a fixed amount of delay to her outgoing packets while accepting all incoming packets. This cheat allows her to receive updates faster than she is sending them, granting her the ability to respond to game events quicker. For example, Eve has a 10ms connection to Alice, but artificially adds a 140ms delay on outgoing updates. Thus, Eve can react to updates from Alice 10ms later, while Alice will not see Eve's updates until 140ms after they occurred. The effect of disproportionate latency was examined in [3], which showed that the added latency directly affects the ability to win in some games.

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

3) Inconsistency cheat: In the inconsistency cheat, Eve sends her 'real' update to every player, except Alice,

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 [4], but in this case we are trying to have an agreement on everyone's game state.

4) 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.

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

5) 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.

B. Application level cheats

1) Secret revealing: Secret revealing occurs by Eve altering her game client to give her information that would normally not be available. For example, Eve may modify her client so that walls are translucent, giving her the ability to locate enemy players easily.

In the client/server architecture, this cheat can only be prevented by revealing secrets at the last possible moment. In the example above, the server would not reveal Alice's position to Eve until Alice was in Eve's direct line of sight. Unfortunately, this leads to high latency [5].

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

Bots and reflex enhancers are not detectable by a game or preventable by game rules. An extremely skilled player may look like a bot if her accuracy is very high.

C. Game level cheats

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

III. BACKGROUND

Diot, Gautier and Kurose described the first protocol for distributed games [6], [7] 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 [8]. 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 created the Asynchronous Synchronization (*AS*) protocol[8]. AS extends the basic Lockstep protocol by having players exchange updates only when their actions might intersect. 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.

The AS protocol 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. The AS protocol 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 and AS are vulnerable to the suppressed update cheat–a malicious player can stop sending updates, stopping round progression until other players drop her from the game. Third, Lockstep and AS do not prevent the inconsistency cheat. Last, the AS protocol is vulnerable to collusion. 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 [9] 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 [9] 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 [10]. 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 massively-multiplayer online games due to the routing delay introduced by their architecture [10].

Knutsson et al. also designed a publish/subscribe system [11] using Pastry [12] and Scribe [13]. The virtual world is divided into regions, and players in each region form a group. Each region maps to a multicast group through Scribe so that updates from the players are multicast to the group. Consistency is achieved through the use of *coordinators*. Every object in the game is assigned to a coordinator; therefore, any updates to an object must be sent to the coordinator who resolves any consistency problems. Fault tolerance is achieved through replication. However, their system does not fully take cheating into consideration in its design, but instead relegates it to future work.

In the game industry, very few networked games are fully distributed. One notable exception is Age of Empires (AoE) [14], 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.

IV. 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 was first described in [1].

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 along an application-layer multicast tree). 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 [15].

A. 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 round length is, balancing responsiveness of the game with the ability to play over long network distances.

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,[2] 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:

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

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 $r-1$.

[2]Like the Lockstep protocol, we could also send a hash of the update for this round with the plaintext update for the previous round. Conceptually, the two methods are equivalent—first we *commit* to a move, then we reveal it.

TABLE II
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

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 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 II 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[3]. On the other hand,

[3]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!

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.

B. Proof of Safety and Liveness

We now prove the safety and liveness of NEO by showing that it does not produce an error condition and always progresses. Our assumptions are that players know of each other and any player can authenticate the message of another player (through signatures). An error occurs in our protocol if in round r a player can receive the key for round r from another player before sending out their update for round r. Proving that an error condition never occurs with our protocol also acts as evidence that the lookahead cheat and the fixed-delay cheat cannot be used with our protocol.

Theorem 1: The NEO protocol is safe; therefore, no error condition occurs.

Proof: We proceed by induction on $n \geq 2$, the number of players communicating over NEO. Assume that the current time is t and that the duration of a round is d. Our proposition, $P(n)$, is that for n players, no player can receive a key for the update at time t before $t + d$.

(B): $P(2)$: Let us name the two players as A and B. Without a loss of generality, we assume B will try to cheat. At time t, A sends her update to B. B decides to wait before sending her update until she receives the key from A. By definition of the protocol, A sends her key at $t+d$. Therefore, B cannot receive the key before $t + d$. Furthermore, any update that B sends for time t will ignored by A since it must arrive after $t + d$.

(I): Assume $P(n)$ is true and $n \geq 2$. We now show that $P(n + 1)$ is true. Since our inductive hypothesis states that the n players are unable to receive the key before they send their update for time t, we only need to consider the interactions between one of the original n players, which we will call A and the new player, which we call B. Without loss of generality, we assume B will try to cheat. B waits to send her update until she receives the key from at least one other player. By definition of the protocol, the other players will not send their key until $t + d$. Therefore, B receives a key after $t + d$ and is no longer able to send an update for time t. Thus $P(n + 1)$ is true and therefore NEO is safe.

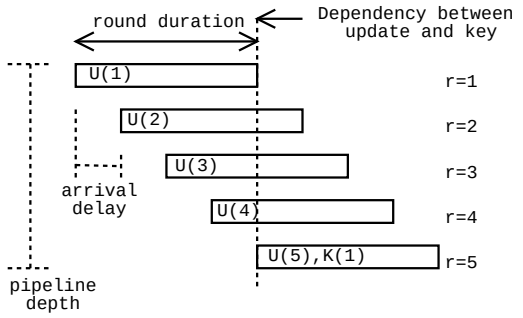

Fig. 1. Pipelining rounds in NEO.

Theorem 2: NEO is always live; thus, rounds advance monotonically with real time.

If we assume that the round duration is d, the current time is t and the start of the game occurred at time s, then we can determine the round number by the function $f(t) = (t - s)/d$. This function increases monotonically with real time. Since NEO does not halt for any purpose, even in the face of inadequate votes, NEO always progresses. Therefore, NEO is live.

C. 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 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 [9]. The pipeline depth is related to the round duration and the round arrival delay, as seen in Figure 1. This relationship can be expressed in the following formula:

$$pipeline\ depth = \frac{round\ duration}{arrival\ delay} \qquad (2)$$

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 1). 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:

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

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

D. Security

Now we explain how NEO prevents the cheats from Section II. We also describe how to prevent an artificial majority.

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 V. 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.

To avoid direct state comparison, hashes of committed moves can act as votes. While this introduces additional bandwidth requirements, these hashes are exactly the digital signatures required by the Byzantine General's problem to prevent inconsistency [4].

Collusion Cheat: NEO addresses collusion at the architectural and protocol levels. First, NEO can adjust the majority value sufficiently high to prevent collusion. Second, each game can be distributed with an individual key that must be registered to play. The registration process can identify players and digitally sign their game keys to prevent a single player from artificially creating multiple identities so that they unfairly control the majority.

V. 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.

To encourage players to participate in round length and rate adjustment, we tie the algorithms into the basic NEO packet. This prevents a player from ignoring a call to vote for a new round length or sending rate because ignoring the call means the player must stop sending packets altogether (or be detected as a non-participant).

A. 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. NEO also measures the delay variance and then each player votes according to their view of the network. 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.

B. 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.

VI. EXPERIMENTS

In this section, we describe our simulation methodology and experimental results. We show that NEO significantly outperforms Lockstep with regard to playout latency. We then show that we can adjust both the round length and rate of NEO to adapt to network conditions.

A. Methodology

We use the *ns-2* simulator [16] to simulate NEO and Lockstep. *ns-2* is a packet-level simulator that allows us to study the effects of dropped or delayed packets on the performance of the protocols.

Baughman and Levine state that the Lockstep protocol uses reliable transport, though the actual implementation details are not included [8]. As a result, we have implemented the Lockstep protocol as best we could. Originally, we used TCP as the underlying transport protocol, but quickly discovered its abysmal performance as with Lockstep in the face of packet loss. Instead, we have re-implemented Lockstep on top of UDP using a simple reliable transport protocol to maximize Lockstep's performance.

We optimize Lockstep's stop-and-wait mechanism by allowing any player that has received all hashed updates to immediately send out their plain-text update. As a further optimization, we allow the last player that sends the plain-text update to include the hash of their next update, as the authors suggest [8].

We do not implement event scoping in Lockstep[4] because we assume that all players are close enough in the virtual world that event scoping would not be useful. Furthermore, the addition of event scoping might obscure the performance characteristics of the protocols. We leave studying the effects of event scoping as future work.

[4]Baughman and Levine use the term *asynchronous synchronization* for event scoping [8].

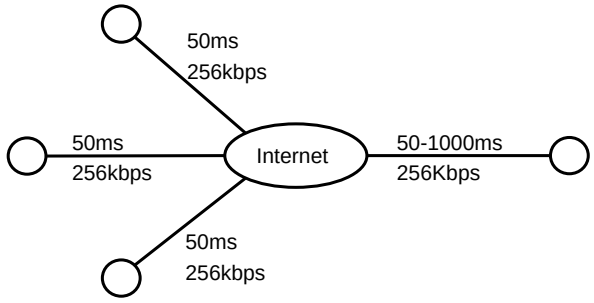

Fig. 2. Simulation Topology

NEO is implemented as described in the Section IV. NEO's configurable parameters are discussed in the appropriate sections below.

We use a simple star topology in order to measure the performance of NEO and Lockstep as seen in Figure 2. While more complicated network topologies could have been used, we argue that by using this simple topology, we can isolate the factors that affect the protocol performance easier without having to consider the effect of an underlying topology.

The primary metric we use to measure performance is *playout latency*. Playout latency is the time from when a player first sends out their update to when the update can be displayed on the screen of the other players. The second metric we use is updates/second. This metric intuitively represents how many updates a player can commit on average per second.

B. NEO vs. Lockstep

1) Maximum Delay: In the first experiment, we compare the playout latency of NEO to Lockstep as delay increases. Obviously, if Lockstep has reasonable performance, then the creation of NEO is superfluous. Our hypothesis, which our simulations verify, is that NEO will maintain a playout latency bounded by its round length while Lockstep's playout latency will increase as the delay of any player increases.

We tested our hypothesis using the simple topology in Figure 2, and increased the latency of a single player from 0ms to 1000ms. This means that the minimum distance between any two players is at least 50ms, while the maximum in our experiments would be 1050ms. In this experiment, we only compare basic NEO to the optimized Lockstep. NEO uses round length of 200ms, a pipeline depth of 1, and does not perform round length or rate adjustments.

Figure 3 plots the playout latency versus the maximum delay of a single player to the other players in the simulation. With Lockstep, when the maximum delay increases to any *single* player, the playout latency for all

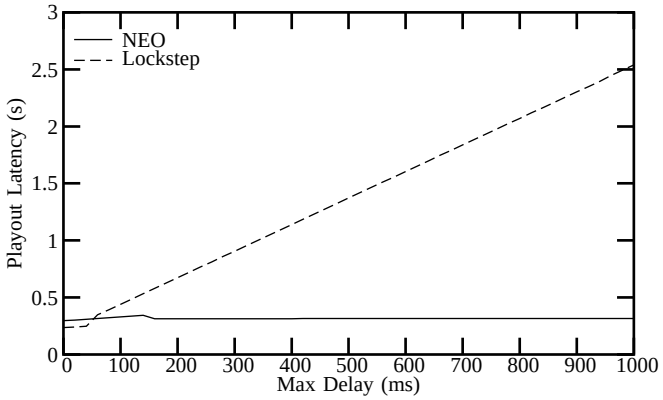

Fig. 3. Average playout latency for all players except for single player with increasing delay.

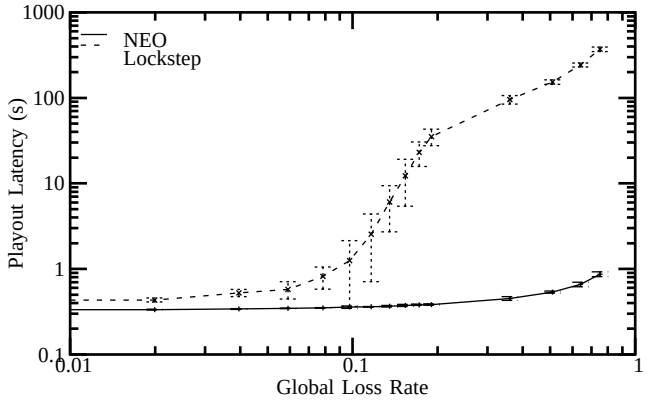

Fig. 4. Average playout latency for all players with global packet loss.

players suffers dramatically. At 0ms, the shortest delay is 50ms, but Lockstep is bound by the longest distance, which is 100ms. Lockstep's optimizations allow it to achieve a playout latency of around 250ms, which is slightly below the predicted 300ms latency. In fact, these optimizations allow Lockstep to achieve a playout latency of about 2.5 times the largest delay between any two players.

With NEO, the playout latency increases only until the single player is beyond the maximum round length. At this point, the playout latency drops back down to its expected value, about 2 times the round length. Once the single player is beyond 200ms, her moves are dropped by the other players.

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

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

2) Global Packet Loss: In the second experiment, we compare the playout latency of NEO to Lockstep when packet loss occurs. Our hypothesis is that Lockstep will suffer from high playout latency with increased packet loss, while NEO will only suffer from a poor playout latency when packet loss is significant.

For this experiment, we first examine how the protocols behave when all links experience the same packet loss, and we vary the loss rate (generated by a uniform distribution) from .01 to .75. Our simulator is configured with the same topology previously used except that one player has a 100ms delay to the Internet. We compare the playout latencies of the optimized Lockstep to basic NEO with a round length of 250ms. Each experiment is run a minimum of 5 repetitions, with new random seeds, until the standard error is below 5% or to a maximum of 100 repetitions.

Figure 4 shows the results of this experiment. We plot on a log-log scale so that our graph can more clearly display packet loss under 10%. The graph also uses error bars to display the 95% confidence interval on the average taken.

Lockstep demonstrates two trends. First, it consistently has a worse playout latency than NEO. Second, its round lengths fluctuate wildly under packet loss. This trend is due to the use of reliable transport which must determine when a packet has actually been lost.

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

3) Packet Loss - Single Player: We extend the previous experiment by examining how Lockstep and NEO perform when only a single player is experiencing packet loss. Rarely will an entire group experience the same

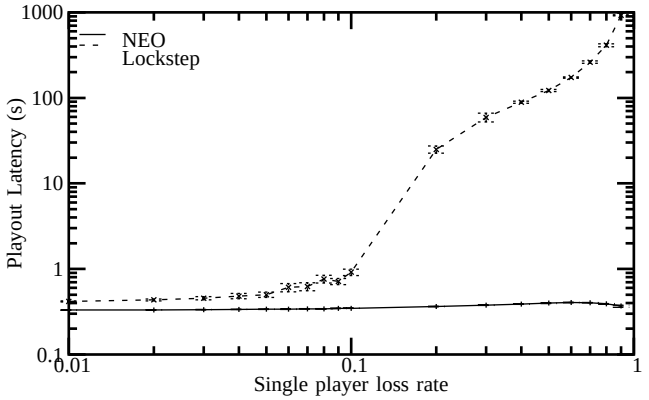

Fig. 5. Average playout latency for all players except single player with increasing packet loss.

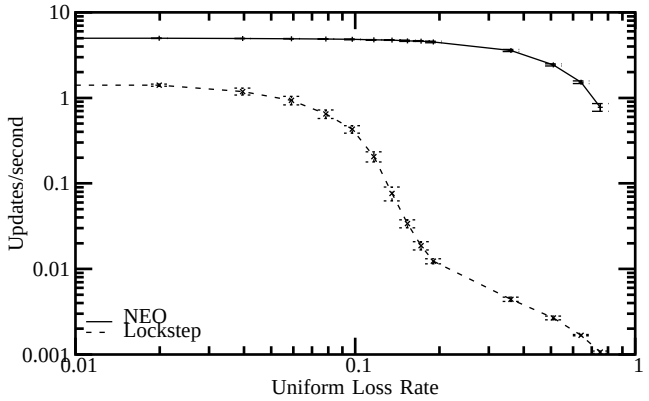

Fig. 6. Average move rate for all players except single player with increasing packet loss.

packet loss rate. Thus, in this experiment, we use the same parameters as the previous experiment, except that we vary the packet loss of the player who is 100ms from the Internet from 1% to 90%. We hypothesize that Lockstep will suffer from packet loss in a manner similar to the previous experiment, but that NEO players will be unaffected by a single player with a poor connection. Figure 5 shows the results from our experiment.

Lockstep shows the familiar trend of being adversely affected by packet loss, but this time a single player affects the entire group of players. Examining Figures 4 and 5 reveals that packet loss by a single player or an entire group is detrimental to the playout latency. Any player experiencing a brief, but modest, 10% packet loss will render the game unplayable for everyone.

On the other hand, NEO performs extremely well. None of the other players are affected by the single player experiencing high packet loss.

4) Updates/Second: In our final comparison between Lockstep and NEO, we attempt to answer the question of how the playout latency translates into updates per second for each player as packet loss increases. We repeat the previous experiment with global packet loss and Figure 6 shows our results.

Our results show that while the playout latency for low packet loss or low delay may not seem large, it translates into a disparagingly low update rate for players. At slightly over 1 update/second, Lockstep would not be useable for most interactive, real-time games. In comparison, NEO performs well under packet loss and delay.

C. Round Length Adjustment

Beyond a comparison of NEO to Lockstep, we have examine the performance of NEO's round length and rate adjustment algorithms. Players using the NEO protocol need to not only agree on how to adjust the round length and sending rate, but also when they will change those parameters. We hypothesize and show that NEO players can agree on a new round length relatively quickly, even though they must exchange messages and agree in a distributed fashion.

For this simulation, we use the same topology, but three players are at 50ms from the Internet and one player is 225ms from the Internet (for a maximum delay of 275ms between the farthest players). We use a global loss rate of 10% to explore round adjustment in the face of packet loss. To agree on a round length, players vote to increase, decrease, or remain at the same duration. When a majority of players vote to increase, the round length is doubled. When they vote to decrease, the round length is reduced by 20ms. All NEO players maintain a pipeline depth of 1 and begin with a round length of 100ms. The maximum round length is set at 300ms.

Figures 7 shows the result of the experiment for round length adjustment. The figure only shows the first 5 seconds since the round length stabilizes after three seconds. The straight, horizontal line represents the maximum delay between any two players–which is the target goal of our adjustment algorithm. The lines overlap for players 2-4. Player 1 initially takes longer to increase her round length to 200ms because of her distance to the other players. However, beyond this, the players are able to quickly agree on the new round length and adjust it appropriately. The speed at which NEO is able to adjust to the new round length is quite satisfactory.

As discussed before, NEO uses an exponential weighted average when measuring delay between players and takes the variance of the average into consideration when deciding how to adjust the round length. Using an exponential weighted average and calculating the variance prevents NEO from prematurely adjusting the round length.

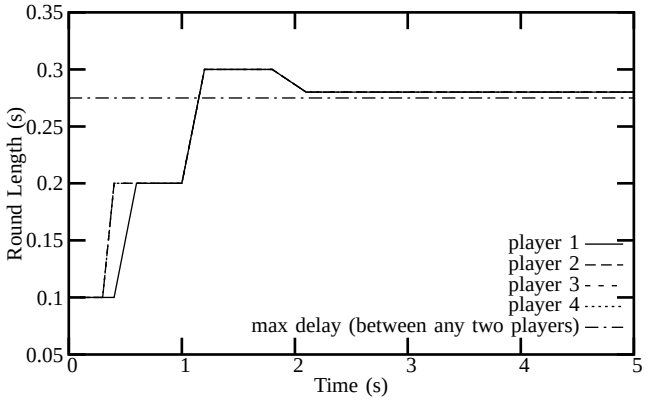

Fig. 7. Round length adjustment algorithm discovering maximum latency between players.

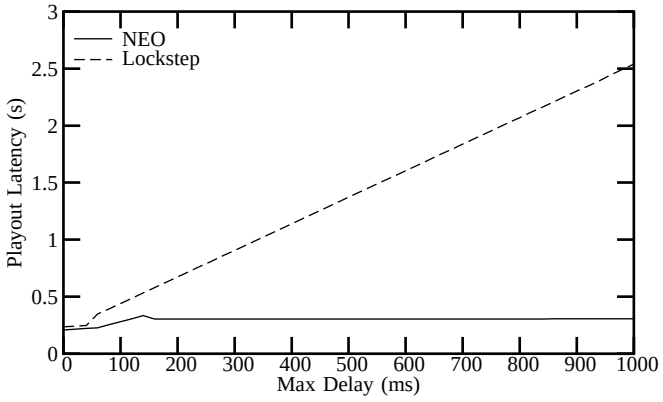

Fig. 8. Average playout latency as delay increases with NEO's round length adjustment algorithm.

With round length adjustment, we can re-examine the comparison between NEO and Lockstep. Looking closely at Figure 3 reveals that Lockstep actually performs better when the delay between players is small. This is because the playout latency in NEO is guided by the round length. Once we add round length adjustment to NEO, its playout latency surpasses Lockstep at all data points as shown in Figure 8.

D. Rate Adjustment

Besides reacting to changing delays on the Internet, NEO needs to react to packet loss as an indication of congestion so that it can back off on its sending rate. By adjusting the pipeline depth (analogous to the congestion window in TCP), we can slow down or speed up the rate that NEO injects packets into the network. We hypothesize that NEO will adjust the sending rate globally in response to packet loss, though the speed at which it adjusts will be determined by the round length.

For this simulation, we placed everyone at 50ms from the Internet. We introduced an artificial 10% global packet loss rate. While an artificial 10% packet loss is not sufficient to demonstrate proper rate adjustment, these

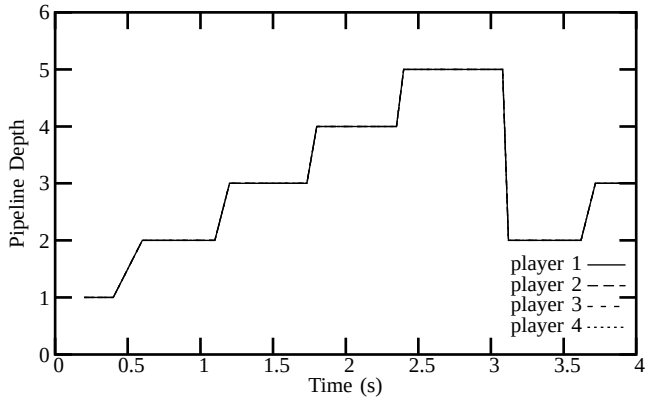

Fig. 9. Rate adjustment algorithm with global 10% packet loss.

results are a preliminary evaluation of rate adjustment in NEO. A complete design of rate adjustment for games, including testing against live traffic, congestion control, and fairness is left as future work.

Initially, each player has a pipeline depth of 1 and measures the sending and receiving rate to each other player. The round length is fixed at 200ms and the maximum pipeline depth is set to 5.

Figure 9 shows the result of this experiment. All player lines in the graph overlap. Initially, all players are not experiencing significant packet loss so they begin to increase their sending rate. However, as their pipeline rate reaches 5, they have an increased probability for packet loss and therefore their measured rate is lower than their maximum rate. Thus, at 3 seconds into the simulation, they cut their sending rate in half. Without packet loss, the rate will ramp up to its maximum and flatten out for the duration of the simulation (not shown due to space constraints).

This experiment shows that rate adjustment reacts to lost packets quickly and appropriately cuts the sending rate for all players. Each adjustment takes about two round lengths in delay.

VII. CONCLUSION

The problem of trust and event ordering in communication protocols is a pervasive problem in distributed systems. In networked online games, this problem manifests itself through cheating players who try to subvert the communication protocol to gain an unfair advantage over other players. Networked games also introduce the problem of trying to maintain real-time, interactiveness between players. Taken in isolation, we can solve either of these problems at the exclusion of the other. For maximum security, we can ignore the interactiveness component of games. For maximum performance, we can ignore the problem of cheating.

The NEO protocol solves both of these challenging problems. NEO handles the problem of security by encrypting updates and then revealing the keys at a later time. It also handles the problem of interactive communication by bounding the maximum latency of any update from a player by the current round length.

The NEO protocol is provably secure against common protocol level cheats. Our simulations show that it adapts to changes in delay on the network by increasing or decreasing its round length. It also reacts to congestion by changing its sending rate. Furthermore, the majority of players in NEO will always experience the best protocol performance because NEO adapts to the requirements of the majority of players. Majority voting also prevents lost packets from unnecessarily being recovered and keeps NEO viable even in the face of high packet loss.

These properties of NEO lead us to believe that NEO is a valuable protocol for peer-to-peer, real-time, interactive games. Our simulations lead us to believe that a real-world implementation will perform well. As future work, we plan on developing a congestion-controlled rate adjustment algorithm for NEO that is friendly to TCP. We also plan on implementing NEO with a real peer-to-peer game to evaluate user experiences with its performance over the Internet.

REFERENCES

[1] C. GauthierDickey, D. Zappala, V. Lo, and J. Marr, "Low Latency and Cheat-proof Event Ordering for Peer-to-Peer Games," in *ACM NOSSDAV*, June 2004.

[2] C. GauthierDickey, D. Zappala, and V. Lo, "A Fully Distributed Architecture for Massively Multiplayer Online Games," in *ACM NetGames Workshop*, August 2004.

[3] J. Nichols and M. Claypool, "The Effects of Latency on Online Madden NFL Football," in *ACM NOSSDAV*, June 2004.

[4] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[5] K. Li, S. Ding, and D. McCreary, "Analysis of State Exposure Control to Prevent Cheating in Online Games," in *ACM NOSSDAV*, June 2004.

[6] C. Diot and L. Gautier, "A Distributed Architecture for Multiplayer Interactive Applications on the Internet," *IEEE Networks magazine*, vol. 13, no. 4, July/August 1999.

[7] L. Gautier, C. Diot, and J. Kurose, "End-to-end transmission control mechanisms for multiparty interactive applications on the internet," in *IEEE Infocom*, 1999.

[8] N. E. Baughman and B. N. Levine, "Cheat-proof Playout for Centralized and Distributed Online Games," in *INFOCOM*, 2001, pp. 104–113.

[9] E. Cronin, B. Filstrup, and S. Jamin, "Cheat-Proofing Dead Reckoned Multiplayer Games," in *International Conference on Application and Development of Computer Games*, January 2003.

[10] A. R. Bharambe, S. Rao, and S. Seshan, "Mercury: A Scalable Publish-Subscribe System for Internet Games," in *Proceedings of the First Workshop on Network and System Support for Games.*, April 2002.

[11] B. Knutsson, H. Lu, W. Xu, and B. Hopkins, "Peer-to-Peer Support for Massively Multiplayer Games," in *IEEE Infocom*, March 2004.

[12] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. Springer-Verlag, 2001, pp. 329–350.

[13] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron, "SCRIBE: A large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in Communications (JSAC)*, 2002.

[14] P. Bettner and M. Terrano, "1500 archers on a 28.8: Network programming in the Age of Empires and beyond," in *Game Developers Conference*, March 2001.

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

[16] "The Network Simulator - ns-2," http://www.isi.edu/nsnam/ns/.