Cheat-Proof Event Ordering for Peer-to-Peer Games

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

Abstract—This paper presents a peer-to-peer communication protocol for fully distributed multi-player games that prevents four possible network level cheats, while maintaining consistency and low latency. We explain the network level cheats and why any are fatal to any peer-to-peer game. Our protocol is an important building block for large-scale, peer-to-peer, multi-player games.

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, 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 massive investment in resources required by a client/server architecture. Furthermore, this architecture allows games to overcome the bottleneck of server-only computation by harnessing the processing power of the players’ machines.

The fundamental problem with a distributed game is trust. 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 orders events while ensuring trust. NEO is designed to be cheat-proof at the network level and is the first protocol to prevent all four common network-level cheats. Moreover, NEO also provides much lower latency than previous event ordering protocols. Previous protocols are bound 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 okay to be slow to some players, as long as most players get your updates in a timely fashion.

In this paper we describe the NEO protocol and explain how it prevents cheating. We further improve on its performance by introducing several enhancements. We then briefly illustrate how NEO can be used as a building block for a P2P game architecture.

II. A TAXONOMY OF CHEATING

We first present a taxonomy of cheating in order to clarify both the terminology and the need for protocols that prevent network-level cheats. Cheats result from security flaws, and we can taxonomize them into categories based on the layer at which they occur: game, application, or network.

Game level cheats occur by breaking the rules of the game, perhaps because the rules are not correctly specified. For example, Eve uses some combination of objects which grants her powers she should not normally have. 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 enemy players. Network level cheats occur by modifying packets or changing when they are sent. We have identified four network-level cheats that affect

One local game developer states that bandwidth requirement for their massively-multiplayer game is equivalent to the city of Eugene’s telephone bandwidth.
distributed multiplayer games, and describe them in the context of the cheating player Eve:

1) Lookahead cheat: In this cheat, Eve purposely adds a fixed amount of delay to outgoing packets, allowing her to receive updates faster than she is sending them. This gives her the ability to see other players’ moves before moving herself. This cheat is not possible with a client/server architecture, as the server orders events as it receives them.

2) Timestamp cheat: In a distributed system that uses timestamps to maintain consistency, each event is timestamped by the sender to ensure consistency for the event time across all players. However, Eve can cheat by waiting to receive an update from Alice and then sending 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 provides a total ordering on events, telling each client when events occur.

3) Suppressed Update cheat: In this cheat, Eve suppresses updates to other players. As a result, the other players do not know where she is exactly or what actions she has performed, giving her the ability to ‘hide’. Eve sends updates only right before she would be disconnected for packet loss.

4) Inconsistency cheat: In the inconsistency cheat, Eve sends different updates to different players in an attempt to cause problems with the game. This cheat is not beneficial to Eve and is used for malicious purposes only. Any distributed system must be able to detect or prevent inconsistencies in the state of the game. The use of native multicast prevents this cheat.

III. MOTIVATION

Previous research on communication protocols does not address both the real-time and security requirements of distributed games. Distributed games need to prevent network-level cheating, but must also provide low latency to ensure real-time game play.

Diot, Gautier and Kurose described the first protocol for distributed games in [1] and [2] 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 and preventing the inconsistency cheat. No other cheats are prevented.

At the other end of the spectrum, Baughman and Levine presented the lockstep protocol to address the problem of cheating with dead-reckoning [3]. Lockstep works by dividing game time into rounds, during which players reliably send a cryptographic hash of their move to all other players. Once every player has received the hash, the plain-text move is then reliably sent to all players. The game proceeds to the next round only after the hash and the move have been received by all players.

Lockstep is a major advance in distributed protocols because it is provably secure against the lookahead and timestamp cheats. The drawback of lockstep is that its playout latency, which is the time from when an update is sent out to when the update can be displayed to other players, is unacceptably high for real-time games. The use of reliable transport bounds its playout latency at 3 times the maximum delay between any two players, assuming no packets are lost.

Cronin et al. designed the sliding pipeline protocol[4] in order to improve the lockstep protocol. An adaptive pipeline is added 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 sliding pipeline reduces jitter and dead-reckoning, it still has the same playout latency as lockstep. In terms of security, the protocol prevents the lookahead cheat, but allows a player to use the suppressed update cheat. While their protocol uses the adaptive pipeline to help detect this cheat, it can falsely label someone with an increased delay as a cheater. Furthermore, a cheater can use the suppressed update every other round and not be detected.

In the game industry, very few networked games are fully distributed. One notable exception is Age of Empires (AoE) [5], [6], 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 addresses some of the same issues, but all participants are trusted, so the DIS protocols do not attempt to prevent network-level cheating.
IV. NEW EVENT ORDERING PROTOCOL

We designed the New-Event Ordering (NEO) protocol to prevent network-level cheating while maintaining low latency in a fully-distributed, peer-to-peer game. Like the lockstep protocol [3], NEO uses encrypted updates and only releases the key for an update at a later time. However, to provide better performance, NEO divides game time into rounds and prevents players from sending late updates. This bounds the time during which a player may issue an update, thus making it possible to release a key for each update in a timely fashion. 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.

A. The Basic Protocol

For simplicity, we start with a basic protocol that prevents only the lookahead and timestamp cheats. We later extend this protocol to address the other two network-level cheats. In this protocol, time is broken into equivalent 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 bound 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.

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

$$M'_A = E(S_A(U^r_A)), K^{r-1}_A, S_A(V^{r-1})$$

(1)

In this message, $E(x)$ is an encrypted $x$, $S_A(x)$ represents $A$’s signature on $x$, $U^r_A$ is the update from player $A$ for round $r$, $K^{r-1}_A$ is $A$’s key for the update from round $r - 1$, and $V^{r-1}$ is the bit vector of votes for messages 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 vote 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 I 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 update, 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. 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

<table>
<thead>
<tr>
<th>Player</th>
<th>Bit-vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1 1 0 1 0</td>
</tr>
<tr>
<td>B</td>
<td>0 1 0 1 0</td>
</tr>
<tr>
<td>C</td>
<td>1 1 1 0 0</td>
</tr>
<tr>
<td>D</td>
<td>1 1 1 1 1</td>
</tr>
<tr>
<td>E</td>
<td>packet lost</td>
</tr>
</tbody>
</table>

Voting tally 3 4 2 3 1

If a majority are not receiving updates, then the game is unplayable. But this holds true for any game, distributed or not!
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 lookahead 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.

B. Pipelining 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 size increases the time that 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. 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:

$$\text{pipeline depth} = \frac{\text{round duration}}{\text{arrival delay}}$$  \hspace{1cm} (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 $t=80\text{ms}$ and the round duration is $120\text{ms}$, then the key must not be sent until $t=200\text{ms}$. We can now generalize Equation 1 using the round duration $d$ and round time $r$ for player A in the following equation:

$$M^r_A = E(S_A(U^r_A)), K^{r-d}_A, S_A(V^{r-d})$$  \hspace{1cm} (3)

Now, as the sending rate of updates increases, both the responsiveness and the visual smoothness of the game increases.

C. Handling Other Cheats

The basic protocol prevents the lookahead and fixed delay cheats, but does not cope with the other two cheats. If multicast is used, the update suppression and inconsistency cheat are not possible with NEO. Any player suppressing updates will find their moves ‘ignored’ by the other players. Furthermore, a player is unable to send different packets to different players using multicast.

For unicast, players can prevent the update suppression cheat with local rate adjustment, described in section V. To prevent the inconsistency cheat under unicast, players periodically compare the state of the game with each other, using a new message that includes a hash of the game state at a given time. Differences in the hash at a given time signify that players must reconcile their state. Since players must digitally sign their packets, the group can determine which player cheated and respond appropriately.

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.

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 void 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.

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. To determine when to skip updates, a player keeps track of the loss rate on a per-player basis. When packets are dropped from one player to another, the player cuts her sending rate in half to that player. For each round duration that passes without a loss, the player increases her sending rate by one, up to the maximum global sending rate. Local rate adjustment has the added benefit that it prevents the update suppression cheat. Any player which attempts to suppress packets to another player will find that the other player will immediately begin to suppress messages in return.

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.

VI. TOWARDS A PEER-TO-PEER GAME ARCHITECTURE

A distributed MMOG consists of three components. First, the communication architecture determines how players send messages to each other. NEO is a building block of this architecture. Second, the storage component is the mechanism for storing short and long term data. We propose using a distributed hash table, such as CAN [8] or Chord [9], and using security mechanisms to prevent cheating with the data. Last, the computation component should distribute the simulation of the game’s artificial intelligence to the players of the game.

We are focusing on the communication architecture. On an abstract level, a game can be broken down into a series of events from players and game systems, such as artificial intelligence. NEO provides event ordering for small groups, but events may have a larger scope than the group that generated it. This requirement means that we need to propagate events from inside a group to other groups, while maintaining the security constraints. In addition to event ordering, group management techniques are needed to build the hierarchies of P2P networks. The group management allows hierarchies to be built from positional information, event scoping, and social structures in a game. Finally, the communication architecture needs to be able to locate players outside of a group with low latency to the majority of the group. This feature is needed to help determine who can be used to farm out computations to. A group might be battling a monster, but cannot be trusted to compute the monster fairly. However, computations can be sent to other players outside of the group, with results compared to ensure their reliability.

REFERENCES