Cheat-Proofing Dead Reckoned Multiplayer Games (Extended Abstract)

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

Keywords— **multiplayer games, cheat-proof protocols, peer-to-peer architectures**

I. Introduction

THE multiplayer game (MPG) market is segmented into a handful of readily identifiable genres, the most popular being *first-person shooters*, *realtime strategy games*, and *role-playing games*. First-person shooters (FPS) such as Quake [11], Half-Life [17], and Unreal Tournament [9] are fast-paced conflicts between up to thirty heavily armed players. Players in realtime strategy (RTS) games like Command & Conquer [19], StarCraft [8], and Age of Empires [18] or role-playing game (RPG) such as Diablo II [7] command tens or hundreds of units in battle against up to seven other players. Persistent virtual worlds such as Ultima Online [2], Everquest [12], and Lineage [14] encompass hundreds of thousands of players at a time (typically served by multiple servers).

Cheating has always been a problem in computer games, and when prizes are involved can become a contractual issue for the game service provider. Here we examine a cheat where players lie about their network latency (and therefore the amount of time they have to react to their opponents) to see into the future and stay ahead of other participants in a game. We focus on cheat-proof protocols in the context of FPSs in a peer-to-peer gaming architecture [6].

While the interactivity of all online multiplayer games is limited by network latency, the effects are most apparent in FPSs. In these games, players must react quickly to their opponents' actions in order to be successful. Players with high-latency connections learn about opponents' actions long after they occur, placing them at a serious disadvantage relative to players with low-latency connections. In RTSs, latencies below 500 ms are generally acceptable [5]; for FPSs, however, acceptable latencies decrease to between 100 and 150 ms [1][4]. FPSs are also more sensitive to the smoothness of the rendered game, determined by how much the latency varies (called *jitter*). If game play is jittery, the ability to accurately aim at a moving target is negatively impacted. In [4], the author discusses the importance of a latency hiding technique known as dead reckoning [10] in the design of the Half-Life game engine, which drives many popular FPSs. Dead reckoning allows play at a client to continue uninterrupted, even if needed moves from another player[1] have not arrived, by predicting the missing moves. The alternative to dead reckoning is to suspend the game until the late moves arrive; this leads to decreased smoothness in the gameplay, which, as described above, damages playability. A problem with the use of dead reckoning is that when moves are incorrectly predicted, inconsistencies can occur. Earlier cheat-proofing protocols [3] do not allow dead reckoning because of these inconsistencies. In [4] however, the author con-

This research is supported in part by the NSF CAREER Award ANI-9734145, the Presidential Early Career Award for Scientists and Engineers (PECASE) 1998, the Alfred P. Sloan Foundation Research Fellowship 2001, and by the United Kingdom Engineering and Physical Sciences Research Council (EPSRC) Grant no. GR/S03577/01, and by equipment grants from Sun Microsystems Inc. and HP/Digital Equipment Corp. Part of this work was completed when Sugih Jamin was visiting the Computer Laboratory at the University of Cambridge.

[1]For ease of exposition, without loss of generality, we assume a two-player scenario in discussing multiplayer gaming.

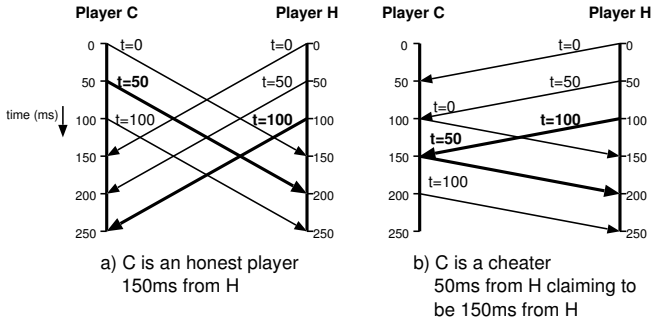


Fig. 1. A sample lookahead cheat.

tends that in the case of an FPS like Half-Life, the negative impact of jerky play due to the avoidance of dead reckoning is greater than the negative impact of any occasional inconsistencies. For this reason, we examine how cheat-proof protocols can be made to coexist with dead reckoning in order to provide both assurances of fairness and acceptable client performance.

II. Background

A. Time Cheats

There are all kinds of cheats used with multiplayer games, from auto-aim robots, to editing files on disk, to making walls invisible [15]. In this paper we examine one particular category of cheats known as *time-cheats*. These cheats give the cheater an unfair advantage by allowing it to see into the future, giving the cheater additional time to react to the other players' moves. Like a number of other cheats, it is very difficult to distinguish between a player employing a time-cheat and one who happens to have high latency or lossy connection and very good luck. Statistical tests can be used to tell when a player is "too lucky" [15], but the cheater needs only to adjust its cheat to be just below this threshold to remain undetected. With detection difficult, if not impossible, prevention becomes the important issue.

The first time-cheat we look at is the *lookahead cheat* [3]. In peer-to-peer games, where each player maintains its own copy of the game state, moves must be timestamped when generated so that they can be executed at the same relative times by each player. Fig. 1b illustrates a lookahead cheat. A cheater (C) exploits client-side timestamping to gain an advantage over an honest player (H). The players are assumed to be 50 ms apart in network latency. In the figure, wall-clock time is represented on the vertical axis, increasing down the page. The diagonal lines represent messages between players, each of which has a timestamp of when it was "sent" (which does not necessarily agree with the wall-clock time when it was generated if the player is cheating). Two of the messages are emphasized in bold. In the figure, C waits until it sees H's move for time 100 before sending out a move timestamped with time 50. This behavior allows player C to see H's moves 50 ms into the future relative to the move it generates, while player H can only see C's moves 150 ms in the past. This results in C having additional time to dodge attacks or otherwise react to H, while H must decide on several of its moves before seeing C's first move. From player H's point of view, the cheater is indistinguishable from an honest player who is 150 ms away (see Fig. 1a), where the two messages cross in the network as expected, giving neither player an advantage. This cheat is prevented by the protocols proposed in [3] and [13] (see Section II-C), but at a significant cost in game latency and jitter.

The second time-cheat we look at is the *suppress-correct cheat* [3] that exploits dead reckoning. This cheat is targeted at systems where the receiver dead reckons up to n missing moves from the other player before assuming that the player has disconnected. In the suppress-correct cheat, a cheater purposefully drops $n-1$ moves and then, having received the other players' last $n-1$ moves, constructs a move based on this information that provides the cheater an advantage. The cheater in this situation is indistinguishable from an honest player with a lossy connection so long as their n^{th} move is plausible. Since dead reckoning was not considered in both [3] and [13], this cheat was also not examined in depth in both papers. In contrast, since we are concerned with dead reckoning in this paper, cheat-proofing suppress-correct cheats is a focus of the protocols presented.

As in previous papers on cheat-proof protocols [3], we assume that cheaters are able to read, modify, inject, or block any game protocol messages between players. In addition, we give cheaters the ability to control how the game interfaces with the cheat-proof protocol, including when moves are submitted and how long the game delays move executions to account for network latency. We do not attempt to protect against other application-level cheats such as those listed in [15].

B. Terminology

We now define the terminology used for describing and evaluating the cheat-proof protocols.

Players issue a *move* once per *frame*. Each frame is a fixed unit of simulation time for all players. A move consists of position change information such as the displacement, change of orientation, weapon firing (position change of the bullet), etc. of a single player. It does not contain event notifications, such as death or teleportation. These events are determined (independently by each client) only after execution of the move. Player i's move issued for frame number n is denoted M_n^i. Moves issued by different players for the same frame enter the game state at the same time. In addition to moves, the cheat-proof protocols also send cryptographically secure one-way *hashes* [16] of each move—as a commitment to that move—before sending the move itself (all hashes, unless otherwise noted, are assumed to be cryptographic). The hash of player i's move for frame n is denoted H_n^i. We assume the use of a transport protocol that ensures reliable, in-order delivery of moves and hashes.

Games are designed to operate at a given *maximum frame rate*, r_{max}; at this rate, the length of each frame is the *minimum frame interval*, $1/r_{max}$. The maximum frame rate controls how often the view on the player's screen is updated. The higher the frame rate, the smoother the motion appears on the screen and the more responsive the game is to player actions. In single-player games, the maximum frame rate is determined solely by the hardware capabilities of the player's computer. In networked multiplayer games however, the maximum frame rate is primarily limited by the rate at which moves can be sent across the network. Quake, for example, typically has a maximum network frame rate of 50 frames per second (a minimum frame interval of 20 ms). There is only one r_{max} for all players in a game because the frame size is the same for each player. Cheat-proof protocols at player i may stall a game for cheat-proofing operations, resulting in a slower actual frame rate, r^i. If the frame rate is not constant, the resulting *game speed jitter* creates choppy game play [18].

Moves are typically scheduled to be executed a fixed *synchronization delay* after they are submitted (instead of immediately) to account for network latency. This synchronization delay allows moves issued at the same time to be executed at the same future time in both the local game and remote games. If a remote move takes longer than the synchronization delay to arrive, the game can either pause, or if dead reckoning is used, the missing move can be extrapolated from earlier moves.

The instantaneous one-way network latency between players i and j is represented by $l_{i,j}$. We assume symmetric latencies; that is, $l_{i,j} = l_{j,i}$ for all i and j. In a game involving more than two players, the highest instantaneous network latency between any two players is l_{max}.

When measuring the performance of a cheat-proof protocol, we look at the *relative game speed*. The relative game speed for player i is the player's actual frame rate r^i divided by the target maximum frame rate r_{max}, as defined above. This ratio is recalculated for each frame since r^i can change from frame to frame. The performance metric r^i/r_{max} shows at what fraction of the optimal speed the game is operating. Variance in the ratio indicates the jitter experienced.

C. Previous Works

Our cheat-proofing protocol for dead reckoned multiplayer games is built upon protocols introduced in [3] and [13]. Due to space constraints, a complete description of these protocols is omitted.

C.1 The Lockstep Protocol

The lockstep (LS) protocol, proposed in [3] requires that all players advance their game clocks synchronously. In the lockstep protocol, player 1 decides on a move M_n^1 for frame n, computes hash H_n^1 and sends it as a commitment to all players. Once all the hashes for a frame have arrived from the other players, each player can send its move to all other players. Players must then wait for moves to arrive from all other players before computing the current frame. After computing the current frame, the player then decides on a move for the next frame and computes a new hash. If less than the minimum frame interval has passed since it sent the previous hash, the player must wait before sending out the next hash. No synchronization delay or dead reckoning of late packets is necessary (or possible) with the lockstep protocol, since a player must always stop and wait until the previous moves of all players have arrived before deciding on the next move.

The biggest drawback of the lockstep protocol to an FPS is that the speed of the game depends on the network latency of the slowest link, l_{max} [3]. The frame interval can be calculated using Eq. 1:

$$\frac{1}{r^i} = \max\left\{2 \cdot l_{max}, \frac{1}{r_{max}}\right\}. \quad (1)$$

If l_{max} is greater than half the minimum frame generation interval, $1/r_{max}$, game speed for all players will be slower than the target game speed. Any jitter in l_{max} will then be reflected in the frame interval, resulting in choppy game speed.

C.2 The Pipelined Lockstep Protocol

The lockstep protocol assumes that players need to see opponents' previous moves before deciding on the next move. However, if a synchronization delay is used, the opponents' moves for a given frame will not be rendered until some time after the decision for that frame has already been made. The pipelined lockstep (PLS) protocol, proposed in [13], takes advantage of the synchronization delay by sending several hashes before the corresponding opponents' hashes are received.

Under the PLS protocol, player i can send out its move M_n^i as soon as it has received hash H_n^j from every other player j. However, since game speed is not affected as long as other players receive M_n^i and can slide their pipelines forward before they have to compute H_{n+2p}^j, where p is the pipeline size, we assume player i sends H_{n+2}^i with M_n^i, H_{n+3}^i with M_{n+1}^i, and so on.[2] The PLS protocol is able to provide the same cheat-proof guarantee as the lockstep protocol, in that no player can see another player's move i before committing to its own move i.

The performance of the pipelined lockstep protocol depends on the pipeline size. Eq. 2 describes the frame interval of the pipeline protocol, where p represents the pipeline size.

$$\frac{1}{r^i} = \max\left\{\frac{l_{max}}{p}, \frac{1}{r_{max}}\right\}, \quad (2)$$

$$p_{opt} = l_{max} \cdot r_{max}. \quad (3)$$

If the maximum frame rate is fixed, the pipeline size p determines the protocol's performance. If p is set too high, there is less jitter but the overall delay increases. If p is set too low ($p = 1$ is simply the lockstep protocol), the frame rate can be exposed to network jitter. Eq. 2 shows that $p \geq l_{max} \cdot r_{max}$ makes l_{max}/p no larger than the minimum frame interval. Setting p larger than necessary simply increases delay, hence we set p_{opt} to the smallest value that satisfies the maximum frame rate (Eq. 3).

[2]Player i could put itself in a disadvantagous situation similar to the one depicted in Fig. 2 if it sends its moves earlier than required by the protocol.

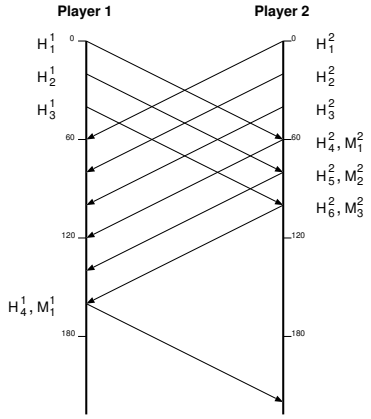


Fig. 2. Cheating under the Pipelined Lockstep protocol.

As under the lockstep protocol, if there is increased delay and moves are late, since the PLS protocol does not allow for dead reckoning, the game must stall.

With the PLS protocol, a player places itself at a disadvantage by deciding its current move based on a past view of the world. However, it does this under the assumption that every other player is doing the same, and therefore it is at no greater disadvantage than any other player. If it is assumed that a cheater can control its own synchronization delay and the point at which it commits to a move, the PLS protocol is, unfortunately, unable to guarantee this level of cheat-proofing.

Fig. 2 shows a cheat where player 1 is able to decide on moves based on full knowledge of all of player 2's previous moves while player 2 (an honest player) decides on its move based only on player 1's moves up to six moves back ($2p$ frames, assuming $p = 3$). At time 60, player 1 receives H_1^2, and is expected to decide upon its next move and send H_4^1, M_1^1 to all players. Player 2 does as is expected and sends H_4^2, M_1^2. As the next three hashes arrive, player 2 continues to commit to moves and send out hashes; player 1 continues to do nothing. After player 2 sends H_6^2, M_3^2, it cannot send any new messages because it has reached the end of its pipeline and has not received H_4^1 that would allow it to slide its pipeline forward. At time 160, all of player 2's outstanding messages have arrived. Player 1 then looks at M_1^2, M_2^2, and M_3^2 to decide on M_4^1, to which it commits and sends out H_4^1. At this point, the two players will advance one move at a time, with player 1 always at a five move advantage when deciding its next move. With larger pipeline sizes, this may provide player 1 with a considerable advantage: a pipeline of 10 would mean player 2 would move based

on where player 1 was located 20 frames ago. We call this the *late-commit* cheat.

III. Adaptive Protocols

The above cheat-proof protocols fall short of the performance requirements for FPSs, in terms of both latency and jitter. In this section we propose a new adaptive protocol that adjusts to network conditions and address the late-commit cheat exposed in the PLS protocol. Our new protocol takes advantage of dead reckoning in order to mask the jitter and provide a constant frame rate to the players while still remaining cheat-proof.

A. The Adaptive Pipeline Protocol

As seen in Section II-C.2, using the pipelined lockstep protocol has two main problems: (1) efficiently computing a good p_{opt} (Eq. 3) that will last an entire protocol session, and (2) preventing cheaters from waiting before committing to moves (the late-commit cheat). The solution to both these problems lies in knowing the value of l_{max} and adjusting the protocol to adapt to it. Unfortunately, l_{max} is not likely to remain constant throughout a game. In this section we present the adaptive pipeline (AP) protocol that addresses these problems by making p a dynamic parameter that adjusts with l_{max} to keep the actual frame rate as close as possible to the maximum frame rate. In addition, by taking l_{max} into account in determining when it is safe to commit to a move, the cheat presented in the previous section can be detected and bounded to a single pipeline. Like the lockstep and pipelined lockstep protocols, the AP protocol does *not* allow for the use of dead reckoning to compensate for late moves.

A.1 Efficient Computation of p_{opt}

To address the first problem of efficiently computing p_{opt}, the adaptive pipeline protocol requires every player i to continually measure the latency $l_{i,j}$ to every other player j. We assume that this measurement is done by passive monitoring of when hashes and moves are sent as part of the normal running of the AP protocol. For player i, l^i_{max} is the largest latency between itself and any other player, given by Eq. 4. The maximum of all l^i_{max}'s for a particular frame is l_{max} (Eq. 5).

$$l^i_{max} = \max\{l_{i,j} | j \in Players\}, \quad (4)$$

$$l_{max} = \max\{l^i_{max} | i \in Players\}. \quad (5)$$

begin ADAPTIVE-PIPELINE-PROTOCOL

RECEIVE H^j_{n-p} and update l^j_{max} for all j

$l_{max} \leftarrow$ MAX($\{l^j_{max} | j \in Players\}$)
$old\_p \leftarrow p$
$p \leftarrow \lceil l_{max} \cdot r_{max} \rceil$
$l^i_{max} \leftarrow$ MAX($\{l_{i,j} | j \in Players\}$)

if $p > old\_p$
  // grow pipeline size
  SEND ($M^i_{n-old\_p}, H^i_n, H^i_{n+1}, ..., H^i_{n+p-old\_p}, l^i_{max}$)
else if $p < old\_p$
  // shrink pipeline size
  SEND ($M^i_{n-old\_p}$)
else
  // same pipeline size
  SEND($M^i_{n-old\_p}, H^i_n, l^i_{max}$)

$n \leftarrow n + 1$

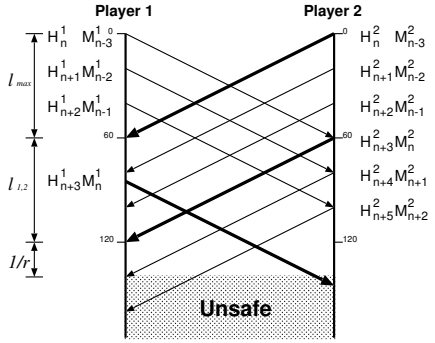
repeat

Fig. 3. The Adaptive Pipeline protocol at player i for frame n.

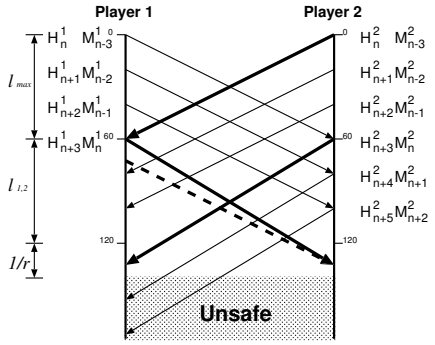
To faciliate the distributed computation of l_{max}, each game message carries l^i_{max} in addition to a move and a hash.

Since the global l_{max} is computed after each frame, it is possible for each player to adjust its pipeline size either by sending more hashes (to increase p) or by sending the moves but no new hashes (to decrease p) over the next several frames. The adaptive pipeline protocol operation for a frame is given in Fig. 3. New l^i_{max}'s are sent only with hashes because only the receipt of hashes marks frame advancement, when the l_{max} can be recomputed. When l_{max} increases and the pipeline size grows, multiple hashes are sent in the same message. Computation of l_{max} associated with each of these hashes uses the single l^i_{max} included with the message.

The frame rate for the AP protocol is still given by Eq. 2 since the calculated l_{max} is only an estimate of the global maximum latency when the message is sent. If l_{max} does not change significantly between when the $l_{i,j}$ measurements are made and the message is sent, the frame interval is near optimal. The disadvantage to the AP protocol's dynamic sizing of p is that although latency is reduced, each time p changes, the synchronization delay for the game changes, which introduces jitter. Since dead reckoning is not supported by the AP protocol, it too is not well suited for FPSs due to this jitter. Instead, we use it as a stepping point to the Sliding Pipeline protocol.



a) Player 1 is a cheater



b) $l_{1,2}$ is larger than estimated

Fig. 4. Detecting cheating under the Pipelined Lockstep protocol with $p = 3$, $l_{max} = 60$ and $l_{1,2} = 60$.

A.2 Detecting and Bounding Late-Commit Cheat

The second problem faced by the PLS protocol is the late-commit cheat illustrated in Fig. 2. Under the AP protocol, knowing l_{max} and $l_{i,j}$ allows us to detect when a player may be using this cheat. This detection has no false negatives (if a player is cheating it *will* be detected), but players who experience a sharp increase in latency could falsely be labeled as cheaters. After describing how the detection works, we will show that it does not require trusted knowledge of l_{max} and $l_{i,j}$ and discuss ways that a game designer can deal with potential late-commit cheaters.

As long as player i receives player j's move M_n^j before $l_{i,j} + 1/r^i$ time from when it sent out M_n^i, it is impossible that player j could have received more moves before sending its move M_n^j. Fig. 4a shows how this detection can be done. We focus on the move for frame n. As discussed in Section III-A, in the pipelined protocols, moves are assumed to be sent out as late as possible: the first move of a pipeline is sent out at the start of the next pipeline. In the figure, after H_n^i is sent out at time 0, both players must wait l_{max} before scheduling M_n^i with H_{n+3}^i Player 2 notes when move M_n^2 was sent, and when it received move M_n^1 from Player 1. If the gap is less than $l_{1,2} + 1/r^2$, Player 2 knows that Player 1 could not have waited for any more of Player 2's moves before deciding on M_{n+3}^1 and sending its hash. The shaded area in the figure represents the *unsafe region*. If M_n^1 is received after time 140, it is possible that Player 1 received extra moves from Player 2 before deciding on M_{n+3}^1.

Fig. 4b shows how this cheat detection can still account for some variance in $l_{i,j}$. At time 60, $l_{1,2}$ increases from 60 to 70. Player 1's message arrives later than player 2 was expecting it (time 130 instead of time 120). The dashed line is the projection of Player 1's move's transmittal based on what Player 2 thinks $l_{1,2}$ is. Even if $l_{1,2}$ were still 60, Player 1 would have sent the move before M_{n+1}^2 had arrived. Therefore, Player 2 knows that Player 1 could not have received any more moves in this period and is not cheating.

A player lying about its l_{max}^i or otherwise causing the measurement of $l_{i,j}$ to be larger than its actual value simply increases the estimated l_{max}. A larger l_{max} lengthens the pipeline size, and *automatically bounds the negative effect of the false information to a single pipeline*. Thus the effect of a false l_{max} on the cheat-proofing ability of the AP protocol is bounded to a single pipeline size.

A player receiving a move in the unsafe region (see Fig. 4) could mean either the other player is experiencing higher network delay, or the other player is using the late-commit cheat. If l_{max} is estimated based on passive measurements of protocol message exchanges, a cheating player will simply increase the estimated l_{max} and, again, lengthen the pipeline size. Thus the effect of a late-commit cheat is also bounded to a single pipeline size. To continue to benefit from the late-commit cheat, a cheating player would have to continually increase its wait time. The game designer should certainly be suspicious of increasingly lengthening l_{max}.

Given the possibility that the detection mechanism can return false positive when a player experiences delay spikes, and given that when a player cheats it can gain a one pipeline-full advantage over honest players, game designers using our cheat-proofing protocols should design their games to be robust against occasional unfairness, e.g., by requiring multiple hits to bring down a player, or revert back to the more conservative lockstep protocol for operations that require one-time atomic transaction.

B. The Sliding Pipeline Protocol

Like the AP protocol, the sliding pipeline (SP) protocol dynamically adjusts the pipeline depth to reflect current network conditions. The SP protocol also uses the same cheat-proofing mechanisms used by the AP protocol. The distinction is that SP protocol introduces a *send buffer* to hold moves generated while the pipeline size is adjusted. Combined with optimistic execution using dead reckoning, this allows the game to maintain jitter-free play while still preventing lookahead and suppress-correct cheats.

In earlier cheat-proofing protocols, if a move has not yet arrived when the game is ready to render a frame, it must stall until the move arrives. Similarly, if the protocol is not ready to send the next commitment when the game is ready to issue a move, the game must stall until the protocol can accept the move. In contrast, if the SP protocol is not ready to send a commitment for a move, the move is placed at the end of the send buffer and the game can continue. If a move has not arrived, the game uses optimistic execution to dead reckon the move and resume play. If we assume a practically infinite send buffer, the game never needs to pause and there is no jitter.

The SP protocol is a generalization of the AP protocol. Like the AP protocol, additional latency is minimized and the pipeline size is tuned to maximize frame rate. The SP protocol is able to detect moves sent outside the "safety zone" just as in the AP protocol, preventing the late-commit cheat possible in the PLS protocol. The suppress-correct cheat associated with the use of dead reckoning in [3] also cannot manifest itself under the SP protocol since players are not allowed to drop moves. The SP protocol guarantees both that (1) no cheater sees moves for a frame to which it has not yet committed a move and (2) that no cheater may continually decide on a move with more recent information than a fair player had.

IV. Conclusion

Previously proposed cheat-proof protocols introduce performance penalties that make them unsuitable for latency-sensitive games such as FPSs. The SP protocol we propose in this paper allows for the use of dead reckoning, which hitherto is not supported by cheat-proofing protocols, enabling a multiplayer game to run at its maximum frame rate. The SP protocol, and its intermediary form the AP protocol, also allow for the detection and bounding of the late-commit cheat possible with the PLS protocol.

V. Acknowledgments

We thank Jon Crowcroft for discussions on the cheat proofing protocol presented

References

[1] G. Armitage. Quake3 playing times. http://members.home.net/garmitage/things/quake3-latency-051701.html, May 2001.

[2] Electronic Arts. Ultima Online. http://www.uo.com/.

[3] N.E. Baughman and B.N. Levine. Cheat-proof playout for centralized and distributed online game. In *Proc. of IEEE Infocom 2001*, April 2001.

[4] Y. Bernier. Latency compensating methods in client/server in-game protocol design and optimization. In *Proc. of GDC 2001*, March 2001.

[5] P. Bettner and M. Terrano. 1500 archers on a 28.8: Network programming in Age of Empires and beyond. In *Proc. of GDC 2001*, March 2001.

[6] E. Cronin, B. Filstrup, A.R. Kurc, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. In *Proc. of NetGames 2002*, pages 67–73, 2002.

[7] Blizzard Entertainment. Diablo II. http://www.blizzard.com/diablo2/.

[8] Blizzard Entertainment. Starcraft. http://www.blizzard.com/starcraft/.

[9] Epic Games. Unreal. http://www.unreal.com/.

[10] L. Gautier, C. Diot, and J. Kurose. End-to-end transmission control mechanisms for multiparty interactive applications on the Internet. In *Proc. of IEEE Infocom 1999*, March 1999.

[11] id Software. Quake. http://www.idsoftware.com/quake/.

[12] Verant Interactive. EverQuest. http://www.everquest.com/.

[13] H. Lee, E. Kozlowski, S. Lenker, and S. Jamin. Synchronization and cheat-proofing protocol for real-time mulitplayer games. In *Proc. of Int'l Workshop on Entertainment Computing*, Makuhari, Japan, May 2002. An earlier version was presented at the workshop *Playing with the Future: Development and Directions in Computer Gaming*, Manchester, UK, Apr. 2002.

[14] NCSoft. Lineage—The Blood Pledge. http://www.lineage-us.com/.

[15] M. Pritchard. How to hurt the hackers: The scoop on Internet cheating and how you can combat it. http://www.gamasutra.com/features/20000724/pritchard_01.htm.

[16] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, second edition, 1995.

[17] Valve Software. Half-Life. http://half-life.sierra.com/.

[18] Ensemble Studio. Age of Empires. http://www.microsoft.com/games/age/.

[19] Westwood. Command & Conquer. http://westwood.ea.com/games/ccuniverse/.