# Cheat-Proof Playout for Centralized and Peer-to-Peer Gaming

Nathaniel E. Baughman, Marc Liberatore, and Brian Neil Levine, *Member, IEEE*

***Abstract*—We explore exploits possible for cheating in real-time, multiplayer games for both client-server and serverless architectures. We offer the first formalization of cheating in online games and propose an initial set of strong solutions. We propose a protocol that has provable anti-cheating guarantees, is provably safe and live, but suffers a performance penalty. We then develop an extended version of this protocol, called *asynchronous synchronization*, which avoids the penalty, is serverless, offers provable anti-cheating guarantees, is robust in the presence of packet loss, and provides for significantly increased communication performance. This technique is applicable to common game features as well as clustering and cell-based techniques for massively multiplayer games. Specifically, we provide a zero-knowledge proof protocol so that players are within a specific range of each other, and otherwise have no notion of their distance. Our performance claims are backed by analysis using a simulation based on real game traces.**

***Index Terms*—Gaming, multimedia communication, peer-to-peer networking, security.**

## I. Introduction

THE THREE primary factors that affect the quality of online, massively multiplayer games are: timely playout of real-time interaction; scalability of communication and game architectures to large numbers of users; and the prevention or detection of cheating players. Although cheating abounds in current game play on the Internet, there is little or no real security to prevent cheating in online games. Previous work in prevention and detection of cheaters in games has been largely relegated to mental poker [1] and similar games.

Modifications to games are often sophisticated open-source, community efforts at reverse-engineering the games. For example, the Decal project offers to expose information about game play normally hidden from players.[1] Such modifications are simple to download and use.

Online, real-time, strategy games, first-person shooters, and massively-multiplayer virtual worlds all rely on a client-server-based architecture for execution of game play. This architecture offers a single point of game coordination, but it creates a bottleneck of processing and signaling as the size of and number of players in online worlds increases. Moving to a peer-to-peer (i.e., serverless) architecture increases scalability and performance, but complicates player interaction and increases the already troubling potential for cheating extant in centralized approaches.

In this paper, we make fair and cheat-proof interaction a necessary condition of correct game play. We offer several contributions that make cheat-proof interaction among players feasible for the first time in a peer-to-peer network.

First, we show that the common synchronization techniques used to preserve the real-time quality of online games, generally called *dead reckoning*, are detrimental to correct game play and can create irresolvable conditions for multiplayer coordination. We show that cheating players are indistinguishable from noncheating players under such techniques. For example, with dead reckoning, a player can cheat simply by delaying until other players announce their moves for the next few frames. Then, the cheating player can alter their strategy and announce moves for recent past frames as if the packets were delayed in the network. Other players will accept the late packets.

Second, we propose a simple protocol for peer-to-peer game play, called *Lockstep*, and prove cheating is not possible. We extend the protocol with a new technique that improves performance but preserves security. We call the extended protocol *asynchronous synchronization* (AS). We evaluate AS and Lockstep with traces from a multiplayer game and show that AS improves performance over Lockstep significantly. For example, in a simulation with 75 players, 95% of frames were stalled at least 10 ms under Lockstep, while only 40% of frames were stalled at least 10 ms under AS.

Third, we extend the scalability of AS with a cell-based protocol. Cell-based protocols allow players to limit most of their interactions to opponents within their local region, thereby reducing network traffic and decreasing playout delays. Our proposed cell-based protocol allows players to determine if they are in the same cell, or near the border of two cells. Importantly, players that are not in the same cell or near a border do not learn the respective location of the other player. This prevents the exposure of too much information that can assist in player strategy.

There are several differences between the preliminary publication of this work [2] and this version. First, the performance evaluations discussed in Section VI use a increased number of simulated players, and the section now includes a discussion of scalability and jitter. Second, Section VII presents a previously unpublished protocol for cell-based gaming. The new protocol removes a critical problem with the preliminary work: players

Manuscript received April 8, 2004; revised September 29, 2006; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor K. Almeroth. This work was supported in part by National Science Foundation Award CNS-0133055. This paper was previously published in part in the Proceedings of the IEEE INFOCOM 2001.

The authors are with the Department of Computer Science, University of Massachusetts, Amherst, MA 01003 USA (e-mail: baughman@prisms.cs.umass.edu; liberato@cs.umass.edu; brian@cs.umass.edu).

Digital Object Identifier 10.1109/TNET.2006.886289

[1]Available: http://decal.acdev.org

one pixel apart but in adjacent cells could not determine they were so close. This prevented running the AS protocol between players in adjacent cells. The new protocol ensures that players will know if they are within half of a cell-width of one another, even if in different cells. We did not introduce additional exchanges between players in the new protocol.

The remainder of this paper is organized around those goals. We present background information on game architectures in Section III. Section IV considers the potential for cheating during game synchronization. Section V presents new protocols for cheat prevention. Section VI presents an analysis of the network performance by simulation of the protocols. Section VII looks how our techniques may be combined with clustering and cell-based techniques for scaling to massively multiplayer scenarios. Section VIII concludes.

## II. Background

*Dead reckoning* is a technique that compensates for variable communication latency and loss across a network by allowing a client to guess the state of another player when updates are missing based on the last known vectors. Dead reckoning is a part of the Distributed Interactive Simulation (DIS) and High-Level Architecture (HLA) standards [3], [4], and is commonly used by researchers and developers [5]–[11]. In its simplest form, the predicted position of a player is equal to the previous position plus the velocity times the elapsed time. Singhal and Cheriton [9] have refined this basic formulation. Diot, Gautier, and Kurose [7], [8] have evaluated the performance of *bucket synchronization* with dead reckoning in a simple, distributed game called MiMaze. Bucket synchronization provisions a series of buckets at each client, one bucket per discrete time unit in the game. Each bucket collects state updates sent from each remote player. When it is time to process a bucket, any missing updates are dead-reckoned.

Previous work places dead reckoning as a necessary technology for timely game play but does not address the potential for cheating. We show that cheating is possible in a dead reckoning system, and we provide solutions to the cheating problem in later sections of this paper. The next section shows that any form of dead reckoning can lead to irresolvable player interactions if players do not act honestly, especially with a p2p architecture.

Several articles authored by commercial game developers are relevant to our work. For example, Bernier [12] describes how clients compensate for network delay. Pritchard [13] motivates the problem of cheating in games.

Our work is also related to past work on interest management for massively multiplayer games and applications [14]–[19]. Typically, interest grouping is done on the basis of $(x, y)$ grid-coordinates, a natural clustering of interest for virtual environments. Section VII discusses how our work can leverage these techniques.

Lastly, our work is related to parallel simulation techniques [4], [20], [21]. Parallel simulations operate with either *conservative* or *optimistic* event processing. In conservative processing, no entity may be out of synchronization with other entities and therefore no lookahead and processing of events is possible. Optimistic techniques allow for entities to execute events asynchronously, but then must be tolerant of incorrect state or computation during execution. Typically, once it is realized that such incorrect state exists, the computation is undone, or *rolled back*, to the last correct point. This method requires that states are saved. Such techniques are not useful for real-time multiplayer games as it would not be practical to force a human participant to restart to previous points in the game. In this paper, we allow for a limited form of optimistic processing without the need for rollback.

Since the preliminary version of this work was published [2], several works on cheat-proof playout of p2p games followed that have sought to extend our approach or relax its assumptions [22]–[25]. Recently, we have produced our own extension to AS called *Ghost* [26] that manages a p2p cheat-free game for a set of players with heterogeneous network resources. Ghost dynamically creates responsive sub-games based on the delay profiles of players.

## III. Model and Terminology

In this section, we define our model and assumptions regarding the information available to players from software.

### A. Abilities of Players and Cheaters

We grant to *cheaters* the ability to read, insert, modify, and block messages involved with the game protocol application-level signaling. Our goal is to develop techniques to guard against attacks from reading, inserting, modifying, or blocking game messages that provide the players with an advantage in game play. Protecting against attacks on existing transport-layer and network-layer protocols, such as TCP denial-of-service attacks [27], is beyond the scope of this paper. However, such attacks are addressed by our follow on work [26].

We assume application software—i.e., the *client*—is readable by the user and will perform its functions as originally intended. Moreover, any information available to the client is available to the player, e.g., game state or cryptographic keys. This assumption precludes any attempt to employ security by obscurity, which is the predominant model in games today [13]. By allowing the cheater the ability to modify outgoing and incoming messages, we are effectively allowing almost any attack that would require modifying the client itself.

### B. Game State

Games are rule-based, real-time simulations of multiple *entities*, which are in-game objects (e.g., military troops) controlled by a *player*. The game proceeds along a sequence of time units called *frames* that are distinct from *wallclock time*. Each frame, player's make *decisions*, which are actions allowable under game rules that the status of each player's entites. The set of variables that completely describe the status of an entity in the game at a specific frame is called the *entity state*. The set of all entity states that completely describes the status of a game at a specific frame is called the *game state*. Players take exactly one *decision* during each frame. Every frame, the players' decisions must be *resolved* by computing the resulting state according to game rules. The sequence of resolved states observed by each player is the game's *playout*.

A player may control multiple entities in the game, but we may use the terms entity and player interchangeably since it is

the player's decisions that affect game play. Entity actions may be automated, but we still refer to the controlling "player", even though it is not a human user.

Game simulators offer precise game control, but may appear slow to a player if the simulator cannot compute each successive frame quickly enough. A player may also perceive slower game play, in terms of wallclock time, if it must wait for updates from a player across the network.

Under a *client-server* communication architecture, all entity states are computed, maintained, and broadcast by a *centralized* server. The sequence of games states resolved by the server are based on input from clients, who supply player game decisions. Client-server architectures offer a single point of game coordination at the server. In the face of missing client updates, the server uses the entity model to resolve any interactions, which may result in conflicting views of game state between the server and affected client. However, the server has authority over maintaining game state, and the client must accept these discrepancies and conform to the server's view of the game world. The simplicity of maintaining a consistent game state in a client-server architecture has likely contributed to the architecture's popularity among developers, and thus to its prevalence in deployed software.

Under a *peer-to-peer* communication architecture, each client maintains its own entity state, informs other clients of decisions, and resolves any interactions without the use of any centralized authority. In this serverless, *decentralized* model, each client keeps a model of each remote entity. However, when updates from remote clients are missing, the local client must advance its local frame only if the playout will not conflict with every other remote client. We explore this further in the next section and offer a solution.

## IV. Fair Playout

A player cheats by working outside the original client program to read, modify, or block game state to gain an unfair advantage. *Correct* playout of real-time interaction means that the game state is identically perceived by every player. A *fair* game is one where players see events occur as would be expected by games rules and player game actions. The goal of this paper is to achieve correct and fair sequence of games states in a multiplayer, networked game. In terms of the performance, cheat-proof solutions must be scalable to as many players as possible in terms of signaling costs, and delays in advancing game frames must be only minimally perceivable to players.

### A. Irresolvable Interactions

Dead reckoning can be used in client-server or p2p environments [7], [8] and the operation is the same. In the centralized case, interactions are resolved by the server uniformly, but unfairly. If the server uses dead-reckoned state to resolve an interaction, the decision may differ from the expectation of the dead-reckoned player. Thus, a dead-reckoned player may view the server's decision as unfair, since the player's true actions were not used by the server. The resulting discrepancy in playout may cause jumpiness in game play or other artifacts, lowering the quality of the game.

The annoyance of unfair decisions becomes damaging in a p2p architecture: interactions based on dead-reckoned state may corrupt the global game state as seen by each client. We say that an *irresolvable interaction problem* results when dead reckoning is used and interactions are either determined unfairly by a server or potentially incorrectly by a distributed client.

For example, say players $B$ and $C$ dead-reckon player $A$ to take no actions during a particular frame, when from $A$'s point-of-view, she destroyed player $B$. If player $B$ then destroys player $C$ before $A$'s missing actions are resolved, then game play is corrupted. Every player's current game frame could be restored to a state before the interaction, but this is not a fair or practical solution for real-time play—although, it is exactly the approach taken by the High Level Architecture's optimistic time management service [4].

### B. Cheating Under Dead Reckoning

The problem of cheating in multiplayer games is widely recognized, but rarely solved in any definitive way. Many games are designed around the client-server architecture, which provides some implicit security along with centralized control of game state. P2P games are much more prone to cheating, but cheats are possible within a client-server game.

One security flaw under bucket synchronization [8] is what we term the *suppress-correct cheat*, which allows a client to gain an advantage by purposefully dropping update messages. Suppose that under some dead reckoning policy, $n$ buckets are allowed to be dead-reckoned before the player is considered to have lost connectivity and is removed from the game. The value of $n$ thus determines a bound on the maximum delay that is permissible among clients in the game. With such a policy, a cheating player can purposefully drop $n-1$ update packets while playing. The player then uses knowledge of the current game state to construct an update packet for the $n$th bucket that provides some advantage.

A simple example allows a sluggish player, $S$, to chase a more agile player, $A$. $S$ begins pursuit, then drops $n-1$ updates; meanwhile, $A$ dead-reckons $S$'s missing state but cannot confirm where $S$ really is. For the $n$th bucket, $S$ sends a fabricated update that places $S$ on the heels of $A$. As long as $S$ sends plausible updates very $n$th bucket, $A$ cannot confirm that $S$ is cheating or playing fairly; $S$ simply claims to be on a congested, lossy link.

This cheat applies both to client-server as well as distributed games. We can conclude that fair play is indistinguishable from cheating when dead reckoning is used. In the following sections of this paper, we provide strong guarantees of cheat prevention and detection.

## V. Cheat-Proof Game Protocols

Most real-time strategy games require interaction resolution at each discrete game frame. A *stop-and-wait* protocol similar to those used for reliable transport [28] is a simple way to fulfill this requirement for client-server and distributed architectures: before time advances in the game, the state change decisions made by each player must be available. In other words, for a

game at frame $t$, all players stop and wait to receive the announcements of all other players for frame $t+1$, before continuing on to frame $t+2$. Because no dead reckoning is allowed, the suppress-correct cheat is eliminated. Moreover, since all state decisions are known at each turn, all interactions can be resolved correctly and fairly by each client.

Unfortunately, this approach is not sufficient for cheat-free play. A malicious player can decide their action for frame $t+1$ after waiting until some or all other players have announced their respective decisions, which we call the *lookahead cheat*. For example, $B$ may take a lethal shot towards $A$ that could not be defended against in normal human reaction times. However, using the lookahead cheat, player $A$ may have an automated agent that sends the decision to take defensive action in time. When a stop-and-wait-type protocol is employed, players that appear to be slower may actually be implementing a lookahead cheat, which can be serious depending on such game features.

In this section, we present a solution to these cheats called the Lockstep Protocol. We show that the Lockstep protocol is safe and live with a formal proof. We do not intend Lockstep to be used on a network because it's performance is based on the slowest player. It is a stepping stone to our asynchronous synchronization protocol, which improves performance without sacrificing security.

### A. Lockstep Protocol

To counter the lookahead cheat, we propose a stop-and-wait-type protocol with a commitment step. We call this secured version the *Lockstep protocol*.

Suppose frame $t$ is complete. Each player makes a decision for frame $t+1$ and announces a cryptographically secure one-way hash of its decision as a commitment. This announcement includes randomized padding if necessary to avoid recognizable hashed decisions and avoid collisions [1]. Once all players have announced their commitments, players then reveal their decisions in plaintext (including the padding). Clients can easily verify revealed decisions by comparing hashes of plaintext to the previously sent committed value. Because each client has only the current turn information to make its next-turn decision, the lookahead cheat is prevented; waiting is no longer beneficial. As an optimization, the last client is not required to commit its decision if all other clients have already committed theirs; the last player may reveal its decision immediately.

The two-phase commitment and required waiting period for all players in the Lockstep protocol introduces a performance penalty. Although correct playout is preserved with lookahead cheat prevention, the game and all players will run at the speed of the slowest player. The reception of other player's packets is likely to be delayed by current network conditions. Below, we present a synchronization mechanism and protocol that retains the desirable properties of the Lockstep protocol and allows the game to run at a speed independent of all players whenever possible.

*1) Proof of Correctness:* A *safety* and *liveness* proof of the Lockstep protocol shows that it fulfills its requirements by not producing an error condition and always progressing [28]. We make a number of assumptions: there exists a reliable channel between all players; all players know of all other players; players are able to authenticate messages from each other player; and all players wait only a finite time before making decisions and revealing commitments.[2]

*Theorem:* (1) The Lockstep protocol is **safe**: no client ever receives the state of another client before the game rules permit; an error occurs if $A$ knows $B$'s state for frame $t$ before $A$ has committed to events at $t$, where $A$ and $B$ are any two players. (2) The Lockstep protocol is **live**: each player monotonically advances the frame counter with wallclock time within a finite delay.

*Proof:* The safety of the Lockstep protocol follows directly from the protocol specification.[3] Let $F_A$ be equal to the current frame being resolved by an arbitrary player $A$. Initially, $F_A = 0$. As per the protocol description, $A$ announces a hashed version of the decision it has made for $F_A$ once it has received commitments from all other players for the same frame. $A$ will not announce its committed decision for time $F_A+1$ until decisions for time $F_A$ from every other player are revealed, received, and verified against commitments. No player, including $A$, may alter announced events because of the hash commitments. Because $A$ may not advance, there is no possibility that another participant will learn $A$'s decision for a later frame earlier than the one currently being resolved.

To prove liveness, let $t_1$ be the wallclock time at which arbitrary player $A$ starts to resolve frame $F_A$ of the game. Let $t_2$ be the wallclock time at which all players learn the revealed decisions of all other players for frame $F_A$; let $t_2 = \infty$ if this never occurs. Let $t_3$ be the wallclock time at which player $A$ advances to time frame $F_A + 1$; let $t_3 = \infty$ if this never occurs. We will show that $t_1 < t_2 < t_3$ and that $t_3$ is finite.

Assume player $A$ is not the last player to commit. Let $F_A(t)$ equal the value of variable $F_A$ at player $A$ at wallclock time $t$. Let $F_A(t_1) = i$. By definition of the protocol, we know $F_A(t_2) = i$. Because all players wait only a finite time before committing to decisions, and because all communication takes place over a reliable channel, we know the commitment of the last player will be received within a finite time, and therefore, $t_2$ is finite. Because the protocol is safe, we know that the value of $F_A$ is incremented to $i+1$ only at time $t_3$. It is clear from the statement of the algorithm that $F_A(t)$ is a nondecreasing value over time. Because $F_A(t)$ is nondecreasing, $t_2 < t_3$. Because all players reveal commitments within a finite time over a reliable channel, $t_3$ is also finite. □

A similar proof can be constructed if player $A$ is the last player to commit. In that case, it is $A$'s communication that ensures $t_2$ and $t_3$ are finite.

### B. Asynchronous Synchronization

In this section, we present a new synchronization technique with guaranteed fair playout called *asynchronous synchronization* (AS) that relaxes the requirements of the Lockstep protocol method of synchronization by decentralizing the game

[2]All our assumptions can be implemented in practice. For example, players not revealing or committing decisions within a bounded time would be released from game play.

[3]We follow the proof of correctness for stop-and-go protocols given by Bertsekas and Gallager [28, p. 74].

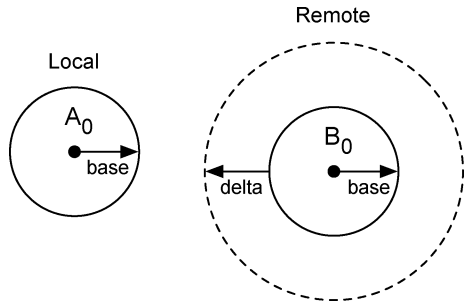


Fig. 1. Base and delta spheres of influence.

clock. Each client advances in time asynchronously from the other clients, but enters into a Lockstep-style mode when interaction is required. Correct playout and fairness are guaranteed. Asynchronous operation of the Lockstep mechanism provides a performance advantage because at times players can advance in game time even without contact from all other players. This relaxed contact requirement may overcome intermittently slow network signaling, packet loss, or slow client processing.

We do not expect AS to be used to allow players with completely different network and client resources to play together. Instead, for this initial design, AS is meant as a technique to isolate the effects of temporarily poor connections between players who play at the same rate for a large majority of time and to reduce the time it takes to resolve interactions. Elsewhere, we have published an extension of AS called *Ghost* [26] that does manage players with heterogeneous resources. Ghost allows each user to set the quality of game they are willing to play and creates the maximum-sized game that satisfies the users' requirements. Notably, AS is the foundation of Ghost.

*1) Spheres of Influence:* Using AS, each player's client keeps track of each other player's advance in game time and space during game play. The area of the game that can possibly be affected by a player in the next turn—and therefore potentially require resolution with other player decisions—is called the player's *sphere of influence* (SOI). We define *influence* as any in-game information that affects a player's decisions, and therefore the outcome of a player's decisions; where *in-game* refers to parts of the game world, as opposed to external knowledge that the player may have, e.g., that a certain opponent typically follows some strategy. It follows that anything outside of a player's current SOI is immaterial to the player's gameplay decisions and resulting events. For example, if a player is not within earshot of a forest, then the player cares not if a falling tree made any sound, nor if it fell. The exact shape and radius of an SOI is game specific.

A geometric SOI is a conservative choice appropriate to almost all real-time games, but a game programmer could exploit knowledge of the game world to tighten the boundary of the SOI.

In AS, each player considers the intersection of two types of SOI. First, a player's own SOI, which indicates a geometric area wherein decisions made contribute to and must be resolved with the player's decisions for the next turn. Second, SOI of remote players, which indicate the areas that can be affected by the other players on their next turn. Accordingly, if two players' SOI do not intersect for a certain turn, their decided events will not affect each other when resolving game state for that turn.

TABLE I
TABLE OF VARIABLES

| | |
|---|---|
| $l$ | Local client |
| $R$ | The set of all remote clients |
| $r$ | A remote client in $R$ |
| $t$ | The current frame at the local client |
| $S_t^h$ | State of client $h$ at frame $t$ |
| $H(S_t^h)$ | Hash of state $S_t^h$ for client $h$ for frame $t$ |
| $p_t^h$ | Potential influence of client $h$ at frame $t$ |

TABLE II
AS AT THE LOCAL PLAYER FOR EACH GAME TURN

1. Compute $S_t^l$
2. Send $H(S_t^l)$
3. Process accepted $H(S_y^r)$ messages that have arrived
4. *foreach* $r \in R$
   Take next $S_y^r$ if any have arrived where $y \leq t$
   Let frame of latest state taken be $x$
   Compute $p_t^l$, and $p_t^r$ dilated from $x$
   *if* $(p_t^l \cap p_t^r = \emptyset)$
   *then* record $l$ is *not waiting* for $r$
   *else if* $H(S_t^r)$ accepted
   *then* $l$ is *not waiting* for $r$
   *else* $l$ is *waiting* for $r$
5. *if* not *waiting* for any $r$
   *then* send $S_t^l$
   resolve any interactions
   finalize and render turn $t$
   advance to turn $(t+1)$

In AS-based games, each client may be making decisions for a different time frame and advancing its time frame independent of other clients; details are described subsequently. Therefore, an SOI is composed of two parts. The *base* SOI is the maximum area that may influence or be influenced on any one turn. The *delta* SOI is the change in influence area that may occur in subsequent turns. Base and delta are represented as radii, as illustrated in Fig. 1, and delta is added to base to compute subsequent turns.[4]

*2) Asynchronous Synchronization Protocol:* Our description of the AS protocol is from the point of view of one client in a peer-to-peer architecture. The protocol can be easily adapted to a centralized architecture.

A formal description of the protocol is given in Tables I and II as it would take place for an arbitrary turn $t$ at a player $l$. If a player has reached turn $t$, then we assume it has already revealed state for turn $t-1$.

For simplicity, we assume in-order, fully reliable delivery of packets. We extend the protocol to out-of-order, unreliable delivery of messages later in this section. Not shown is an initialization phase that occurs before the game begins: every player learns the full set of remote other players, $R$, and starts each remote player, $r \in R$, in lockstep with every other player until the initial positions of the other players are received over the network.

[4]The figures illustrate 2-D play only, but clearly, AS mechanisms exist for 3-D coordinate systems.

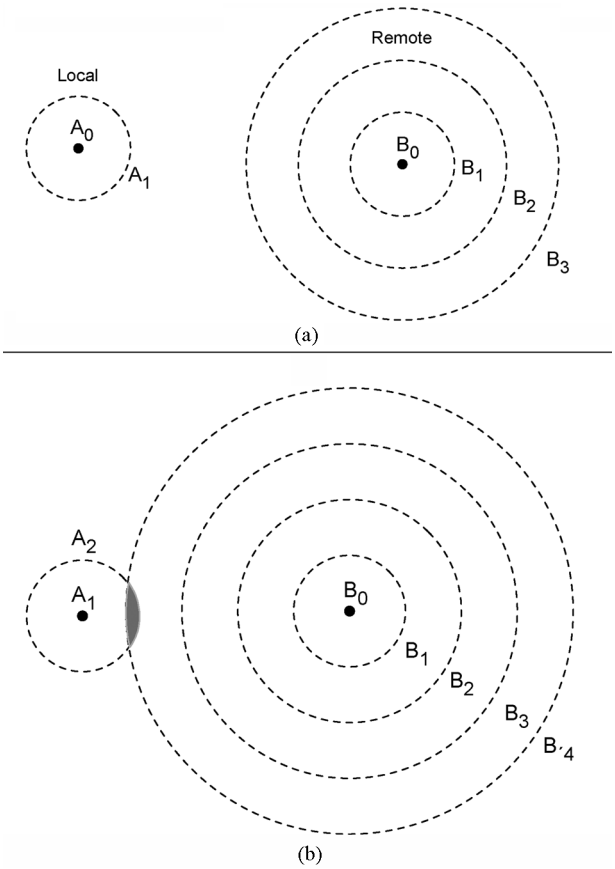


Fig. 2. (a) Dilation to $\mathrm{time} = 3$. (b) Dilation to and intersection at $\mathrm{time} = 4$.

For an arbitrary turn $t$, a player first determines its decision for that turn (Step 1), and second announces the commitment of the decision to all players (Step 2). Third, commitments that are one frame past the last revealed frame of a remote player are accepted (Step 3). Before revealing its commitment, the local player must determine which remote players it is waiting for (Step 4). A remote player is not in the wait state only if there is no intersection with the SOI dilated from the last revealed frame of the remote player, or if a commitment from the remote player has been accepted by the local player.

Each other remote player's SOI is computed using the base radius of the last known position plus a delta radius for each time frame that the local player is ahead of the remote player's last known time frame. If the local client is in the future relative to another player, then the other player's potential to influence the local player's next decision is *dilated* to the local player's next time frame Fig. 2(a). If the local client is not in the future of a remote client, then no dilation is performed. Intersection of the SOI as the local player moves to the next time frame without receiving revealed state from the remote player since frame $t_0$ is illustrated in Fig. 2(b).

Finally, if no remote clients are in the wait state, the local client reveals its state for turn $t$, updates its local entity model of each other player with their last known state, including the remote client's last known time frame (no dead reckoning is performed), and advances to the next turn (Step 5). The protocol then repeats for the next turn.

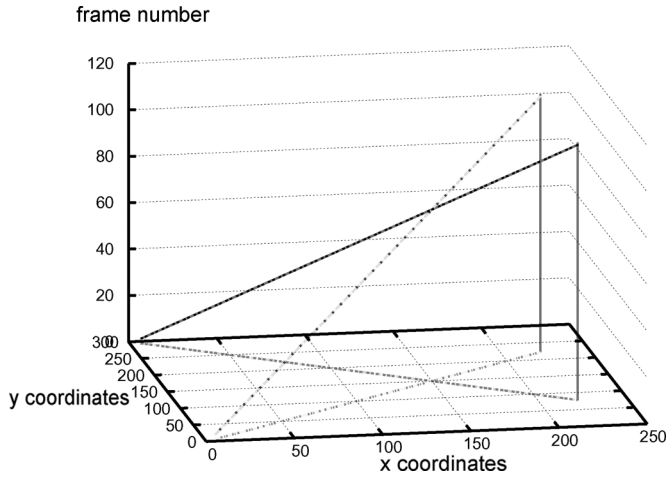


Fig. 3. Player position versus game frames. Top lines: players $A$, $B$, $C$, and $D$. Bottom lines: paths in the $xy$-plane.

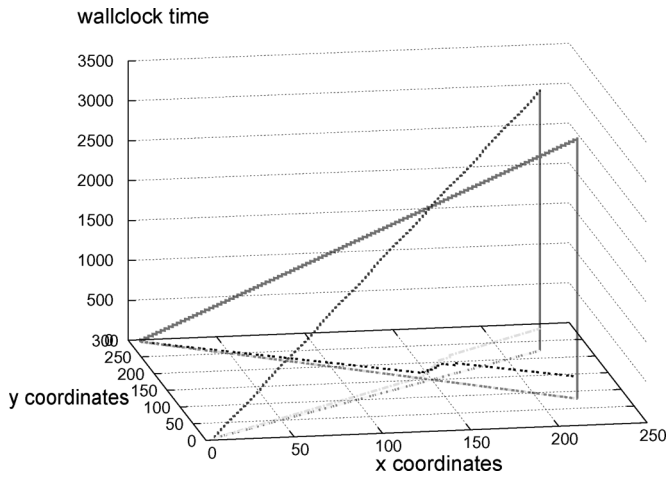


Fig. 4. Player position versus wallclock time. Top lines: players $C$ and $D$. Middle lines: players $A$, and $B$. Bottom lines: paths in the $xy$-plane.

AS allows a client to advance in time at a rate independent of other clients, until there is potential influence from a slower player that might result in interactions that must be resolved.

As an illustrative example, consider the case depicted in Figs. 3 and 4, drawn from simulation results. Fig. 3 shows a pair of players crossing each other in the $xy$-plane of the game. The $z$ axis represents the frame of each player for a corresponding $xy$-coordinate. Both players' paths start at frame zero and end at frame 111. Consider another set of players, $C$ and $D$, that take the same paths in the game, but must proceed in lockstep synchronization. Players $C$ and $D$ have the same frame-versus-$xy$-coordinate graph as players $A$ and $B$. Fig. 4 shows the same paths in the $xy$-plane, but the $z$ axis represents the wallclock time of the players for each coordinate. Players $C$ and $D$ are represented by the two lines advancing slowly in wallclock time (higher in the $z$-plane).

In contrast, players $A$ and $B$ proceed according to the AS algorithm: they may advance in time as quickly as possible, only proceeding in lockstep when their SOI intersect. Fig. 4 shows players $A$ and $B$ only having a sharp increase in slope as the two approach each other in the $xy$-plane. Players $A$ and $B$ need

not wait to hear from the other player to continue with the game otherwise, and therefore are not affected by network delays. In contrast, players $C$ and $D$ must constantly wait for each other, and so network delays affect every moment of the game.

With AS, lookahead cheats are prevented similarly to the Lockstep protocol: by committing to a hash of the next-turn decision until all clients have committed at the same time frame or have revealed past decisions that remove the potential for cheating (i.e., the potential for interaction). AS also eliminates the suppress-correct cheat in the same way as in the Lockstep protocol. The client cannot advance in time until all potential influences for one turn have been resolved. The lookahead cheat might appear to be more serious than in lockstep synchronization: a client may purposely lag behind other clients in order to preview future information. However, a client cannot advance in time past the point where a potential influence is detected, hence cheating is useless as no player would be affected. For example, if a cheater attempts to move to a position outside their SOI, the cheat will be detected as honest clients will see that the cheater has moved to a location outside their dilated SOI.

By the definition of SOI, the future information released by the advancing client is immaterial to any other player's game decisions. However, AS signaling may give a player advance location information about another player, which will not allow cheating of game playout but may possibly alter a player's strategy. We present a solution to this situation for centralized and distributed architectures in Section VII.

The AS protocol also preserves the Lockstep protocol's guaranteed correct and fair playout, since all interactions are resolved with perfect information for each turn in the game. AS also offers a performance increase over Lockstep. In a game using AS, clients may advance their local game clock independent of remote clients when no interactions are possible. We demonstrate such performance gains by simulation in Section VI.

A client in a distributed game using the AS protocol can execute as fast as possible until a potential influence overlap is detected. To preserve a set game play frame rate, designers may impose a maximum game speed. (Simulation results in Section VI make use of a similar type of capped rate.) At best, once an SOI intersection is detected, a client will have to wait for only one update from another player, which will restrict its potential SOI and allow the local client to continue. At worst, the potentially interacting player must catch up in time to a faster client in order to resolve an actual interaction. This worst case occurs, for example, when the lagging player moves directly toward the future position of another player at the maximum delta rate. Otherwise, the past player's dilated SOI will not intersect the future player's SOI, and the future player may continue. As we stated earlier, our protocol Ghost [26] offers are more sophisticated solution to this problem.

*3) AS With Packet Loss:* Although our proof of AS assumed the existence of a reliable channel between all players, this assumption can be relaxed. Simply stated, players can skip missing packets and accept new, out-of-order packets from other players when the missing packets represent state outside a SOI intersection. Missing packets that represent intersection of SOI cannot be dropped or skipped. In fact, packets with very long delay are equivalent to lost packets.

We do not present a new proof here, however, one can be easily constructed by examining if the dilated SOI resulting from missing packets result in an intersection; if they do not, the packet may be skipped. It is clear the protocol has enough information to determine if missing information would possibly result in SOI intersection if eventually received.

We can conclude that AS represents a performance advantage over Lockstep. Rather than contact every player every turn, with AS, players need only contact players that have SOI intersection. In other words, a player 20 SOI radii away need be only heard from after 20 turns to be sure there is no SOI intersection. Other performance benefits are explored in more detail in the next section.

## VI. Performance Analysis

We analyzed the performance of the AS protocol compared to the Lockstep protocol by simulation. We did not compare against dead reckoning techniques as it is clear dead reckoning will perform better but introduces unfair actions for centralized architectures and irresolvable events in p2p architectures, as well as allowing players to cheat.

We took traces of player movements from a representative game, *XPilot* [29], a networked, multiplayer game where players control ships in a two-dimensional space. Accordingly, we cannot claim the results presented in this section are generic. However, our hope is that they are representative.

### A. Methodology

We built a custom simulator that controlled each player in the game based on traces from real XPilot sessions. We configured XPilot to run for about 4000 frames of game play with various numbers of automated players on a 300-by-300 size map,[5] and modified the game to log $xy$-coordinate information to a file. Logging did not begin until all players had joined the game. Our simulator took the logs as input for each player, and each $xy$-coordinate in the log was considered a turn decision taken by each player.

Each unit of simulator time represented 10 ms of wallclock time. Each unit of time in the simulator, players could read from the logs for their next turn and send that turn to other players. However, sending of a decision would be blocked appropriately by the Lockstep or AS protocol according to anti-cheating constraints.

We assumed a star topology, with the server at the center and players at the point of the star. This topology reflects the typical network topology of XPilot when played on the Internet. Each player's network connection had a delay to the center point of the star topology that was drawn each turn from an exponential distribution [30] with a mean of five simulator time units. We also investigated a fixed-unit delay and the results did not provide additional insight, and we do not present them here for the sake of brevity. We did not investigate more biased distributions

[5]This value is the XPilot map size, but the granularity of the player position coordinate system is much finer, on the order of 50 000-by-50 000.

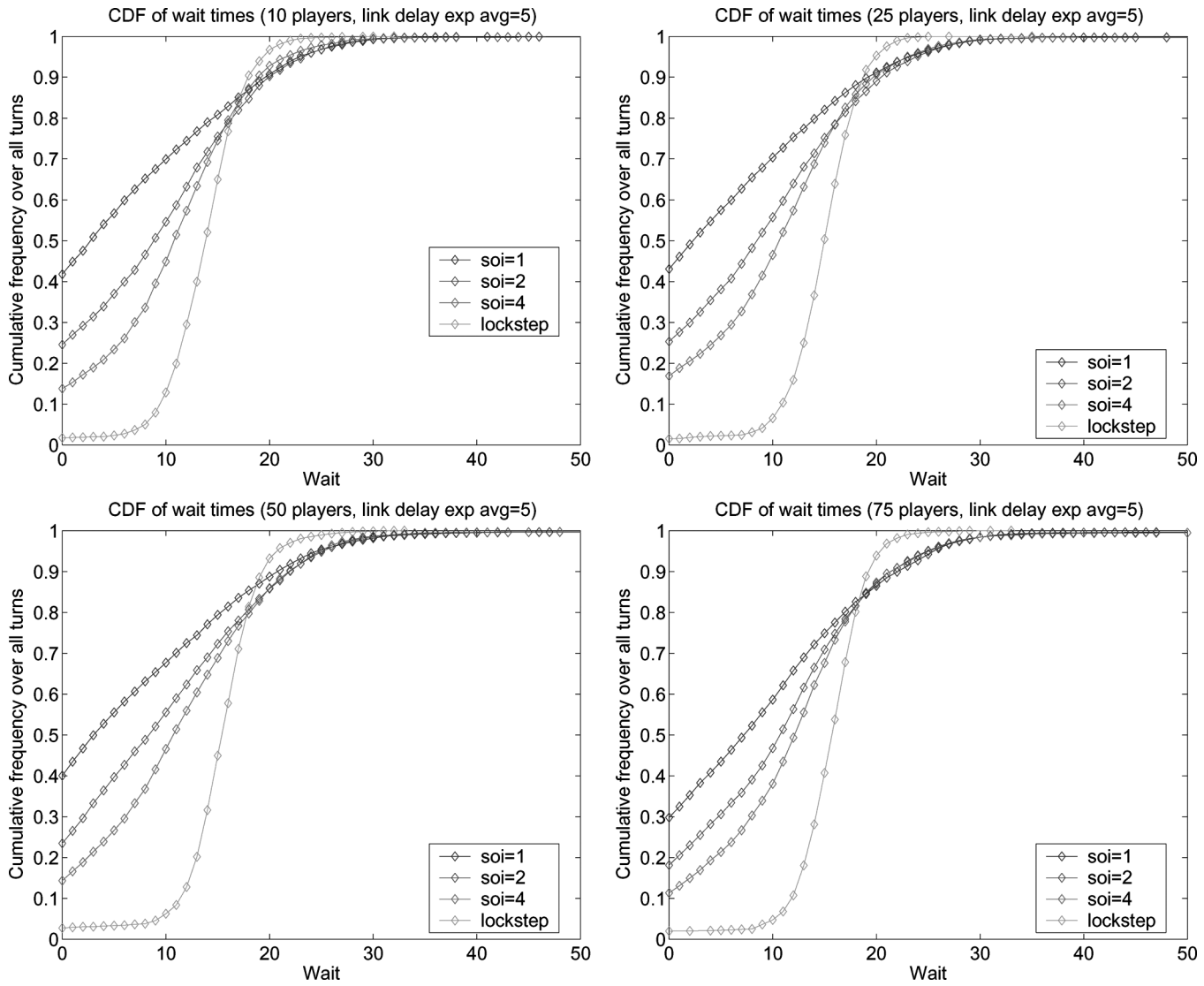


Fig. 5. Cumulative distribution of milliseconds stalled between frames due to lockstep interaction.

of bandwidth, such as a power law, as gaming requires some homogeneity in the quality of network connections among players.

The star topology has two interpretations. The first is that packets were multicast from each player to each other player. The second is that packets were unicast to a nonplaying/nonrefereeing server located at the center of the star, which then immediately and simultaneously unicast the packets to each other player.

Players could not take turns more often than every four units of the simulation so that we were consistent with constraints on human reaction times assumed by previous simulation work [7], [8]; we termed this the *lower cap*. Additionally, players had an *upper cap*: they could not advance in turns more than once for every 10 units of simulator time that had passed. This was to simulate a game running at 10 frames per second, which is consistent with a typical XPilot game.

For example, consider a player at simulator time 30 who just read and sent its decision for game frame 3 to all other players. It must wait 4 steps before reading its next turn (the lower cap), and may not send out the packet until simulator time 40 is reached (the upper cap). However, consider if, due to lockstep constraints, the player was forced to wait until simulator time 51 to send the packet for frame 4. Since four time units had passed since it read frame 4 from the trace, and because time 50 had passed, it could immediately read frame 5 from the trace log and would be able to try to send out the packet immediately (subject to lockstep constraints).

We took traces of 10, 25, 50, and 75 players. Larger numbers of players are possible to simulate—however, our goal was to show quantitative evidence that AS scales better than the Lockstep protocol. Second, it is more likely that massive numbers of players would be designed around a cell-based virtual world, and we provide techniques for cell-based play in Section VII. It is also suitable to consider our simulation as that of one cell in a cell-based or clustered game; this is discussed further in Section VII.

For each trace, our simulator used four different SOI sizes. The smallest SOI usable was set as the maximum distance any player could move in a single turn (an SOI of 1). It is important to note that an SOI of 1 is the size used in practice. The largest

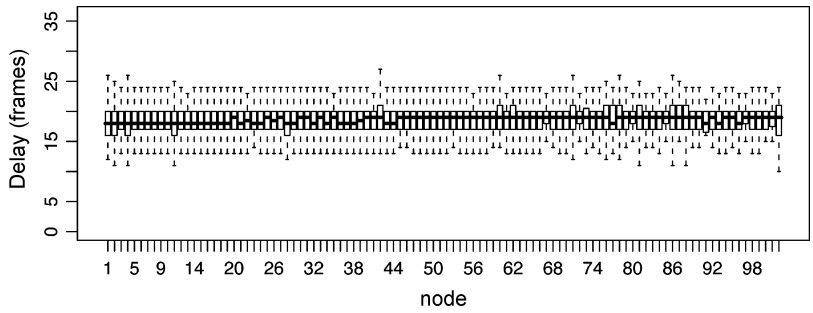


Fig. 6. Per-player wait distributions ("jitter") for one set of simulation parameters. Other simulations showed similar behavior. (SOI = 1, 102 players).

SOI simulated was of infinite size, corresponding exactly to a Lockstep protocol. Strictly for comparison purposes only, we also simulated twice the smallest SOI size (denoted soi = 2 in the graphs), and four times the smallest SOI size (denoted soi = 4).

### B. Results and Discussion

Fig. 5 shows the results for traces of 10, 25, 50, and 75 players. Each graph shows a histogram representing the distribution of time stalled between frames for an average player due to lockstep anti-cheating constraints; i.e., the ms stalled by an average player before each turn's decision could be transmitted over the network as measured from the last turn. Stall time due to the upper and lower caps are not included in these results as they are not involved in AS calculations. involved in AS calculations. Fig. 7 presents the CDF of wait times for AS when the SOI is 1 and the number of players is varied.

The simulation results clearly show the performance advantages of the AS protocol. With the Lockstep protocol, players always wait for the slowest player to send their decision. With the AS protocol, 30%–40% of the turns can be taken without delay due to player coordination while still guaranteeing cheat-proof game play. Even when a player must be stalled, the AS protocol enables players to stall for less time. The simulations show that while AS performance degrades slowly as more players interact, so does the Lockstep protocol, and that AS generally maintains a large advantage. Further, as shown in Fig. 6, the per-player distribution of wait times (i.e., the jitter) is fairly consistent. Fully exploring the effects of AS on jitter is beyond the scope of this paper, and it is discussed more fully in our follow-on work [26].

We expect these results to hold for large numbers of players in large environments. Additionally, in the next section, we present a technique to support distributed *cell-based* game play so that players need only contact other players in their own cell while still following anti-cheating constraints.

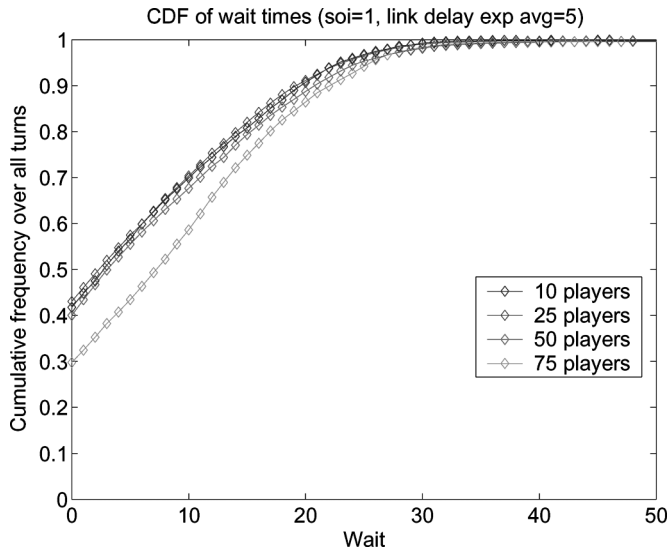


Fig. 7. Cumulative distribution of milliseconds stalled between frames, comparing different player sizes.

## VII. Scaling AS to Massively Multiplayer Games

In order for massively multiplayer games to scale to thousands of participants or more it must be the case that the amount of communication and processing per client must remain low for all entities involved. This is true for server-based and serverless architectures. Rather than employ client- or server-based filtering, one approach commonly used is to cluster participants into separate multicast addresses or separate servers based on geometric position. Accordingly, a virtual playing field may broken into *cells* to increase scalability [14]–[19]. Cells are transparent to the player's view of the game.

The AS technique couples together nicely with cell-based techniques. Cell sizes should be quite a bit larger than the SOI size. A player must only perform AS for the players inside the same cell.

Unfortunately, a cheating player may use information on cell position available in signaling necessary for the correct operation of AS (or dead reckoning or lockstep) in order to learn of an upcoming ambush. While knowledge of the position of a remote ambush or hidden possession may not affect game resolution as discussed in Section IV, it might affect player strategy.

In a client-server architecture, secret information does not present a problem, as the server may be trusted to resolve interactions and advance players without revealing secret information to players.

For distributed architectures, secret information presents a difficult dilemma for AS, dead reckoning, and lockstep approaches. Players must exchange positional information in order to execute the protocols, but the signaling represents an opportunity for cheating by providing advance knowledge of position to players, possibly altering their strategy.

In this section, we present a solution to this problem in the context of the AS protocol. The addition of a hidden positions

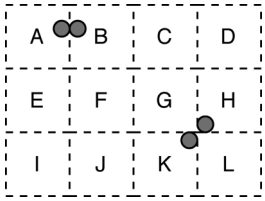


Fig. 8. Players near each other in CBHP [2] separated only by cell borders cannot detect their proximity.

protocol allows a peer-to-peer network game to scale with the number of participants by limiting communication between players distant or isolated from one another in the topography of the game world. In our preliminary work [2], we proposed a protocol that allowed players to discover whether they occupy the same simulation cell, without revealing to each other their current positions. We refer to that protocol as Cell-Based Hidden Position (CBHP).

Briefly stated, in the CBHP protocol, each player contacts every other player upon crossing a cell boundary. Each pair of players determines if their respective current cells' numbers are the same. In this paper, we detail some weaknesses of that design and propose an improved protocol.

### A. Sub-Cell Hidden Positions

In this section, we demonstrate that our previous method of dividing a game world into cells, the CBHP protocol [2], does not always correctly determine player distance, and show a solution to this problem. Further, we show that cell-based protocols can be compromised in a finite number of rounds, and that while this compromise can be delayed and detected, it cannot be prevented.

The CBHP protocol suffers from several limitations:

1. *The CBHP protocol fails to detect player proximity in all cases.*

By dividing a game world into cells, CBHP hidden positions protocol attempts group players according to the topography of the game world. If cells in the game world are topographically isolated by in-game barriers, then the CBHP protocol as described is correct. However, the protocol fails if the cells represent contiguous areas in the game world. Consider a game world divided into cells, as in Fig. 8.

If players are located at the edges of two adjacent cells, e.g., $A$ and $B$, or in the corners of diagonally adjacent cells, e.g., $H$ and $K$, they can be immediately adjacent, yet be in separate cells. This is problematic if the cell based method is used to determine when to enter AS. Players whose SOIs intersect may not be forced into lockstep, and the cheats that AS is designed to prevent may occur. We show a modification to the CBHP protocol that prevents these cheats when adjacent game cells are contiguous areas in the game topography.

2. *The CBHP protocol can be compromised in a number of rounds equal to the number of cells in a game world.*

Suppose players are allowed to engage in the CBHP protocol at any time. There is nothing preventing a malicious player from querying other players many times in quick succession. Each query could claim the malicious player was in a different cell of the game world, thus testing the cell for the presence of other players. By testing all cells in such a manner, the malicious player will determine the cell that each other player is in. If the game rules do not allow a player such knowledge, then this is an exposure. In the worst case, a malicious player would have to query each of the $n$ cells in a game, and thus be able to determine the location of all other players in the game in $O(n)$ queries. We show how the addition of a simple hash function can detect, but not prevent, this sort of cheating.

3. *The CBHP protocol does not provide fine granularity in its Detection of SOI intersections.*

Another limitation of the cell-based method is the lack of granularity in the detection of SOI intersection among players. With a cell-based hidden positions protocol, the only way to increase this granularity is to use a grid with smaller cells. Ideally, a protocol would allow players to know exactly when their SOIs intersect, but would not reveal any other information. In the next, we present a protocol that will allow players to determine more accurate distances from one another at any time.

Additionally, allowing players to join a game in progress is a useful property for a game protocol. We discuss a method by which new players may join an existing game that utilizes a cell-based hidden positions protocol.

### B. Sub-Cell Hidden Positions Protocol

Our modification to the original CBHP protocol of the preliminary version of this paper provides all the required properties of a hidden positions without requiring additional message exchanges. This protocol is based upon dividing cells into subcells, and we refer to as the *sub-cell hidden positions* (SCHP) protocol.

In SCHP, if players are within half a cell width of each other they will be notified; importantly, this is true even if they are in different but adjacent cells. By comparison, in our previous approach, CBHP, players could be on the borders of adjacent cells and not know it (as illustrated in Fig. 8).

*1) Requirements, Terminology, and Notation:* In the SCHP protocol, a game world is divided into $t$ cells, illustrated in Fig. 9. The cells are of size $2n \times 2n$, where $n$ is the minimum distance at which players will be informed of their proximity. Each of these $2n \times 2n$ cells is divided such that there is a square sub-cell $n \times n$ in the center of each, four rectangular area of $n/2 \times n$ and four square areas of $n/2 \times n/2$ around it. These smaller areas will be referred to as sub-cells. These nine sub-cells will be numbered $0 \ldots 8$, and a sub-cell within a cell will be referred to as $a_i$, where $0 \leq i \leq 8$.

A player engages in the SCHP protocol when crossing between cells or sub-cells.

Players will never come closer than within $n$ units of one another before being alerted to this fact by this protocol. This case will occur when players are located on the inner borders of two directly adjacent rectangular sub-cells; this is the case for players $u$ and $v$ in Fig. 9. The furthest apart players can be when detected is $5n\sqrt{2}/2$ units; this is the case for players $s$ and $t$ in Fig. 9.

We are able to achieve this as follows. At any given time in the game, a player is located within some sub-cell in the game world. We will assign the player three *cell designations* based upon this location.

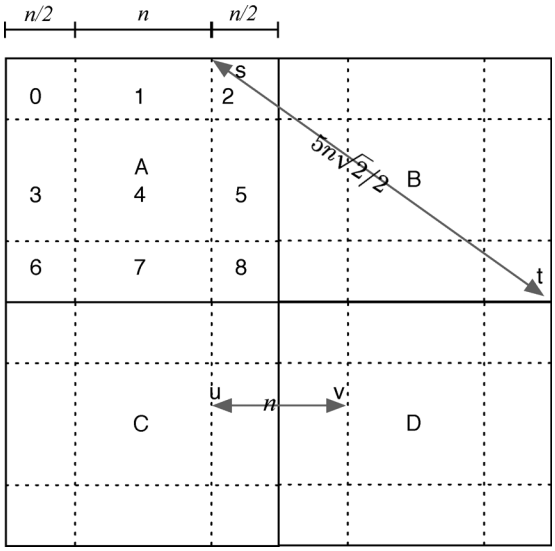


Fig. 9. SCHP divides the game arena into cells each $2n$-by-$2n$; each is divided into nine sub-cells: an $n$-by-$n$ center subcell surrounded by $n$-by-$n$ or $n$-by-$2n$ subcells.

- The first of these cell designations is the cell $1 \leq x \leq t$ in which the player is located.
- The second and third cell designations of the player are determined by the players subcell:
  - — If the player is located in sub-cell $x_6$, $x_7$, or $x_8$, then the next cell designation of the player is that of the cell located below him in the game topography.
  - — If the player is located in sub-cell $x_2$, $x_5$, or $x_8$, then the next cell designation of the player is that of the cell located to the right of him in the game topography.
  - — If either or both of the cell designations are not assigned at this point, then they are chosen at random as a sufficiently large $r > t$.

It is easy to show the bounds on the minimum distances between players in SCHP. Consider the game world in Fig. 9, divided into four cells. If a player is in cell, say, $A$, three possibilities exist, illustrated by considering players in subcells $A_4$, $A_5$, and $A_8$ in Fig. 9:

- Example 1. The player is in the central sub-cell, $A_4$. In this case, the cell designation of the player is $(A, r, r')$, where $r > t$.
- Example 2. The player is in a rectangular sub-cell, such as $A_5$. In this case, the player has a designation of $(A, B, r)$: actual cell $A$, adjacent to, in this case, $B$.
- Example 3. The player is in a corner sub-cell, such as $A_8$. If this is the case, the player has three actual cell designations, one for the current cell of the player, and one for each cell adjacent to the sub cell. In this example, the designations would be $(A, B, C)$.

Below we detail how players can compare their designations without revealing their actual locations.

*2) Protocol:* The SCHP protocol functions with a minimal amount of additional data transferred per message between players as compared to the original CBHP protocol. It does not introduce any additional rounds between players.

As in the original protocol, we require a commutative cryptosystem such as RSA [31] or Pohlig–Hellman [32], so that for any message $M$ we have $\{\{M\}_{K_1}\}_{K_2} = \{\{M\}_{K_2}\}_{K_1}$, where $\{M\}_K$ denotes encryption of a message with key $K$.

*Step 1)* Player $A$ generates three random numbers $R_{A_1}, R_{A_2}$, and $R_{A_3}$ and sends player $B$ a one-way hash of each; $B$ generates three random numbers $R_{B_1}$, $R_{B_2}$, and $R_{B_3}$ and sends $A$ a one-way hash of each. They now have both committed their choices to each other. In addition, both players generate keys, $K_A$ and $K_B$ respectively, to be revealed at the end of the protocol.

$$A : \text{generates } K_A$$
$$B : \text{generates } K_B$$
$$A \rightarrow B : h(R_{A_1}, R_{A_2}, R_{A_3})$$
$$B \rightarrow A : h(R_{B_1}, R_{B_2}, R_{B_3})$$

*Step 2)* Players $A$ and $B$ now exchange their previously committed choices for each $R$.

$$A \rightarrow B : R_{A_1}, R_{A_2}, R_{A_3}$$
$$B \rightarrow A : R_{B_1}, R_{B_2}, R_{B_3}$$

*Step 3)* They independently compute $R_1$, $R_2$, and $R_3$ as the bitwise XOR of each pair $R_{A_i}$ and $R_{B_i}$.

*Step 4a)* $A$ sends $B$ the result of encrypting $(x_i + R_i)$, where each $x_i$ is one of the three current cell designations of $A$, using a random key $K_A$ generated by $A$ and not known to $B$.

$$A : z_i = \{x_i + R_i\}_{K_A}$$
$$A \rightarrow B : z_1, z_2, z_3$$

*Step 4b)* $B$ sends $A$ the result of encrypting $(y_i + R_i)$, where each $y_i$ is one of the three current cell designations of $B$, using a random key $K_B$ generated by $B$ and not known to $A$.

$$B : w_i = \{y_i + R_i\}_{K_B}$$
$$B \rightarrow A : w_1, w_2, w_3$$

*Step 5a)* $A$ sends $B$ the result of encrypting each $w$ that she received in Step 3 using the key $K_A$ that she used in Step 2.

$$A : w'_i = \{w_i\}_{K_A} = \{\{y_i + R_i\}_{K_B}\}_{K_A}$$
$$A \rightarrow B : w'_1, w'_2, w'_3$$

*Step 5b)* $B$ sends $A$ the result of encrypting each $z$ that he received in Step 2 using the key $K_B$ that he used in Step 3.

$$B : z'_i = \{z_i\}_{K_B} = \{\{x_i + R_i\}_{K_A}\}_{K_B}$$
$$B \rightarrow A : z'_1, z'_2, z'_3$$

*Step 6)* $A$ and $B$ both learn whether any $x_i = y_j$ by comparing each $z'_i$ to each $w'_j$: $x_i = y_j$ if and only if $z'_i = w'_j$, which is true by the commutativity of the cryptosystem.

Players can cheat in at least two ways. First, $B$ could ask $A$ to check against every cell in the game sequentially. Second, A could lie about its current position. To avoid both, the keys are eventually revealed. This can be done at the conclusion of the game, or after some period of time after which revealing the cell position is no longer an advantage (i.e., long enough that the player could have moved to a new cell). Revealing the key also

reveals the position from Step 4, which can then be checked for validity against these cheats.

This method of zero-knowledge proof was suggested to us by Mike Atallah of Purdue University; Fagin, Naor, and Winkler offer an excellent summary of related work [33]. The advantage of the approach we present here is it determines success in exactly six exchanges and with no doubt as to the outcome. In comparison, other approaches (see [33]) require tens of rounds of blind signatures and cut-and-choose exchanges to ensure success that only approaches 100% as the number of rounds increases.

*3) Discussion:* The purpose of utilizing a hidden positions protocol is to keep the players out of a slower protocol such as AS until the intersection of their SOIs is imminent. Depending upon the game topography, game rules, and the diameter of the SOI, a value of $n$ may be chosen to minimize the duration of time spent in AS.

At the cost of additional complexity, a grid of a more optimal geometry may be constructed. For example, the SCHP protocol adapted to a hexagonal grid will produce a lower variance between minimum and maximum distance at which player proximity will be detected.

In some games, it may be beneficial to have players relay some information about their proximity to one another. For example, if a player in the SCHP protocol discovers their proximity to another player, he could request from that other player a list of who is near that other player, and enter AS with those players. This may be most relevant in games where players will tend to congregate in the game world.

## VIII. Conclusion

We have made cheat-proof playout a necessary condition for the design of network game communication architectures. We have shown that previous methods of network game communication are exploitable by cheating players. We have proposed the first protocol for providing cheat-proof and fair playout of centralized and distributed network games. To improve upon the performance of this protocol, we have proposed the asynchronous synchronization protocol, which allows for optimistic execution of events without the possibility of conflicting states due to packet loss or the possibility of cheating. Asynchronous synchronization does not require roll back techniques or a centralized server. Our performance analysis shows it significantly improves performance over the Lockstep protocol. Asynchronous synchronization provides implicit robustness in the face of packet loss and allows reduced signaling requirements to be used in combination with cell-based techniques, while always maintaining cheat prevention and detection, allowing for massively multiplayer environments.

## Acknowledgment

The authors are grateful to A. Fagg of the University of Massachusetts, Amherst, and M. Atallah of Purdue University for their helpful insight.

## References

[1] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd ed. New York: Wiley, 1996.

[2] N. E. Baughman and B. N. Levine, "Cheat-proof playout for centralized and distributed online games," in *Proc. IEEE INFOCOM 2001*, pp. 104–113.

[3] *Standard for Information Technology, Protocols for Distributed Interactive Simulation*, ANSI/IEEE Std. 1278-1993, Mar. 1993.

[4] F. Kuhl, R. Weatherly, and J. Dahmann, *Creating Computer Simulation Systems: an Introduction to the High Level Architecture*. Upper Saddle River: Prentice-Hall PTR, 2000.

[5] B. Blau, C. Hughes, M. Michael, and L. Curtis, "Networked virtual environments," in *ACM SIGGRAPH Symp. 3-D Interactive Graphics*, Mar. 1992, pp. 157–160.

[6] E. Berglund and D. Cheriton, "Amaze: a multiplayer computer game," *IEEE Software*, vol. 2, no. 3, pp. 30–39, May 1985.

[7] C. Diot and L. Gautier, "A distributed architecture for multiplayer interactive applications on the Internet," *IEEE Network*, vol. 13, no. 4, pp. 6–15, Jul.–Aug. 1999.

[8] L. Gautier, C. Diot, and J. Kurose, "End-to-end transmission control mechanisms for multiparty interactive applications on the internet," in *Proc. IEEE INFOCOM 1999*, pp. 1470–1479.

[9] S. K. Singhal and D. R. Cheriton, "Exploiting position history for efficient remote rendering in networked virtual reality," *Presence: Teleoperators and Virtual Environments*, vol. 4, no. 2, pp. 169–193, 1995, Also as ACM SIGGRAPH '94 Course 14.

[10] A. Watt and F. Policarpo, *3D Games: Real-time Rendering and Software Technology*. Reading, MA: Addison-Welsey, 2001, ch. 20.

[11] J. Aronson, "Dead reckoning: latency hiding for networked games," *Gamasutra Mag.* Sep. 19, 1997 [Online]. Available: http://www.gamasutra.com/features/19970919/aronson_01.htm

[12] Y. W. Bernier, "Latency compensation techniques methods in client/server in-game protocol design and optimization," presented at the Game Developers Conf. San Jose, CA, Mar. 2000.

[13] M. Pritchard, "How to hurt the hackers," *Game Developer Mag.*, pp. 28–30, Jun. 2000.

[14] D. Van Hook, J. Calvin, and S. Rak, "Approaches to relevance filtering," in *Proc. Workshop on Standards for the Interoperability of Distributed Simulations*, Orlando, FL, Sep. 1994, pp. 367–369.

[15] M. Macedonia, M. Zyda, D. Pratt, D. Brutzman, and P. Barham, "Exploiting reality with multicast groups," *IEEE Comput. Graphics Applicat.*, vol. 15, no. 5, pp. 38–45, Sep. 1995.

[16] S. Rak and D. Van Hook, "Evaluation of grid-based relevance filtering for multicast group assignment," in *Proc. 14th DIS Workshop*, Mar. 1996, pp. 739–747.

[17] E. Léty and T. Turletti, "Issues in designing a communication architecture for large-scale virtual environments," in *Proc. Int. Workshop on Networked Group Communication*, Nov. 1999, pp. 54–71.

[18] K. L. Morse, "An adaptive, distributed algorithm for interest management," Ph.D. dissertation, Univ. California, Irvine, 2000.

[19] B. N. Levine, J. Crowcroft, C. Diot, J. J. Garcia-Luna Aceves, and J. Kurose, "Consideration of receiver interest for IP multicast delivery," in *Proc. IEEE INFOCOM 2000*, pp. 470–479.

[20] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*. New York: Wiley Interscience, Jan. 2000.

[21] ——, "Time management in the high level architecture," *Simulation*, vol. 71, no. 6, pp. 388–400, Dec. 1998.

[22] E. Cronin, B. Filstrup, and S. Jamin, "Cheat-proofing dead reckoned multiplayer games," presented at the 2nd Int. Conf. Application and Development of Computer Games (ADCoG 2003), Hong Kong, Jan. 2003.

[23] H. Lee, E. Kozlowski, S. Lenker, and S. Jamin, "Synchronization and cheat-proofing protocol for real-time mulitplayer games," presented at the IFIP 1st Int. Workshop on Entertainment Computing (IWEC 2002), Makuhari, Japan, May 2002.

[24] C. GauthierDickey, D. Zappala, V. Lo, and J. Marr, "Low latency and cheat-proof event ordering for peer-to-peer games," in *Proc. Int. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2004, pp. 134–139.

[25] C. Chambers, W. C. Feng, W. C. Feng, and D. Saha, "Mitigating information exposure to cheaters in real-time strategy games," in *Proc. Int. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2005, pp. 7–12.

[26] A. St. John and B. N. Levine, "Supporting p2p gaming when players have heterogeneous resources," in *Proc. ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, Jun. 2005.
[27] B. Tjaden, *Fundamentals of Secure Computer Systems*. Wilsonville, OR: Franklin, Beedle and Associates, 2003.
[28] D. Bertsekas and R. Gallager, *Data Networks*. Englewood Cliffs: Prentice-Hall, 1987.
[29] B. Stabell and K. R. Schouten, "The story of xpilot," *ACM Crossroads Student Mag.* Winter, 1996 [Online]. Available: http://www.xpilot.org
[30] J.-C. Bolot, "Characterizing end-to-end packet delay and loss in the Internet," *J. High Speed Networks*, vol. 2, no. 3, pp. 305–323, 1993.
[31] L. Adleman, R. L. Rivest, and A. Shamir, "A method for obtaining digital signature and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.
[32] S. C. Pohlig and M. E. Hellman, "An improved algorithm for computing logarithms over GF(p) and its cryptographic significance," *IEEE Trans. Inf. Theory*, vol. IT-24, pp. 106–110, Jan. 1978.
[33] R. Fagin, M. Naor, and P. Winkler, "Comparing information without leaking it," *Commun. ACM*, vol. 39, no. 5, pp. 77–85, 1996.

**Nathaniel E. Baughman** received the B.S. degree in computer science, with a minor in mathematics, from Ohio Northern University, Ada, OH, in 1998. As an undergraduate, he spent two summer internships with Turbine Entertainment Software Corporation, Westwood, MA. He received the M.S. degree in computer science from the University of Massachusetts, Amherst, in 2000.

He continued his professional career at Syracuse Research Corporation, Syracuse, New York, performing software security research and development, until 2005. Most recently, he is pursuing independent games development with Potential Games in Dover, OH.

**Marc Liberatore** received the B.S. and M.S. degrees in computer science in 2000 and 2003, respectively, from the University of Massachusetts, Amherst, where he is currently working toward the Ph.D. degree in the computer science.

His research interests are in networking and network security.

**Brian Neil Levine** (M'99) received the B.S. degree in applied mathematics and computer science from the State University of New York at Albany in 1994 and the M.S. and Ph.D. degrees in computer engineering from the University of California, Santa Cruz, in 1996 and 1999, respectively.

He is an Associate Professor in the Department of Computer Science at the University of Massachusetts, Amherst, which he joined in 1999. He is also Director of the Center for Academic Excellence in Information Assurance Education at UMass Amherst, as designated by the National Security Agency. His research interests include network privacy and security, mobility, and peer-to-peer networking.

Dr. Levine is currently an associate editor of IEEE/ACM TRANSACTIONS ON NETWORKING. He was co-chair of the 2006 ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV). He was co-chair of the ACM Intl. Workshop on Network Group Communication (NGC) in 2002. He was guest co-editor of IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS special issue on Network Support for Multicast Communications in 2002. He was awarded an NSF CAREER grant in 2001. He has been a member of the ACM since 1999.