Cheat-Proof Playout for Centralized and Distributed Online Games

Nathaniel E. Baughman Brian Neil Levine
baughman@cs.umass.edu brian@cs.umass.edu
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract— **We explore exploits possible for cheating in real-time, multiplayer games for both client-server and distributed, 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, 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 face 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. Our performance claims are backed by analysis using a simulation based on real game traces.**

I. Introduction

Cheating is as old a concept as game playing. For networked games, cheating is closely tied to three major factors that affect the quality of the game: 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.

Online, real-time, strategy games [1], first-person shooters [2], [3], and massively-multiplayer virtual worlds [4], [5] all rely on similar techniques for simulation. Ideally, all players across a network are able to synchronize game events and actions such that player control and interaction is consistent across all player viewpoints. However, games often trade precise control for communication performance gains to preserve the real-time quality of the game playout.

Most popular today are centralized, client-server game architectures, which offer a single point of game coordination, but create a bottleneck of processing and signaling as the size of and number of players in online worlds increases. Moving to a distributed, serverless architecture increases scalability and performance, but complicates player interaction and increases the already troubling potential for cheating extant in centralized approaches.

Cheating abounds in current game play on the Internet, yet there is little or no real security to prevent cheating in online games. Cheats are simple to download and use. In this paper, we make cheat-proof interaction and fair playout of interactions a necessary condition of game communication. We show that the common synchronization techniques used to preserve the real-time quality of online games are detrimental to game play and even create irresolvable situations that destroy the coordination of the game. We uncover the potential for cheating under common synchronization techniques and show that cheating players are indistinguishable from non-cheating players. We then propose a protocol for multiplayer game communication that has anti-cheating guarantees. The protocol is shown to be sufficient to offer precise player interaction coordination, but it is not scalable to large virtual worlds. We then extend the protocol with a new technique for online game synchronization in a way that preserves coordination and security while bettering scalability and performance. Our protocol is also the first to ensure fair playout of events under distributed, serverless architectures, i.e., architectures without a trusted third party.

Section II presents background information on game architectures. Section III considers the potential for cheating during game synchronization. Section IV presents new protocols for cheat prevention. Section V presents an analysis of the performance of the protocols. Section VI looks how our techniques may be combined with clustering and cell-based techniques for scaling to massively multiplayer scenarios. Section VII concludes.

II. Assumptions and Terminology

In this paper, we grant to cheaters the ability to read, insert, modify, and block messages involved with the game protocols. We develop techniques to guard against attacks on such vulnerable application-level signaling. Protecting against attacks on existing transport-layer and network-layer protocols, such as denial-of-service attacks, is beyond the scope of this paper.

We assume application software 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 [6]. Have a compromised client is possible but requires access to and understanding of the original source code or the ability to change the compiled version of the application. Preventing all cheats available because of client modifications are beyond the scope of this initial study and are left for future work. However, the specific techniques in this paper presented for cheat-proof playout are tolerant against modifications to the client.

We refer to the set of information needed to describe the game at any time as *game state*, which is composed of *entity states*. An entity may consist of several in-game objects (e.g., military troops) and is controlled by a *player*. We may refer to a player as a person playing a game as well as the objects that person controls in the game. The partitioning of game state is depicted in Figure 1. Automated players are possible. Players make *decisions*, that is they decide on events that change their own states. When an *interaction* occurs, multiple players' decisions must be

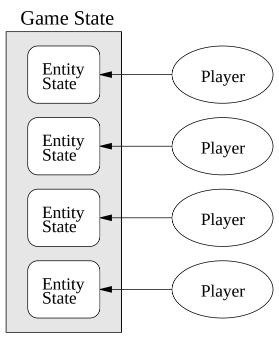

Fig. 1. Game state partitioning

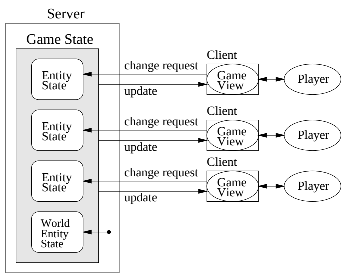

Fig. 2. Centralized-control client-server

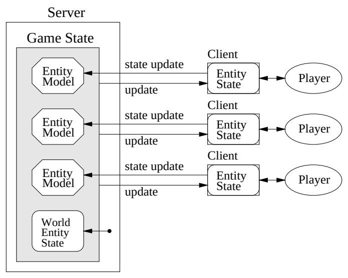

Fig. 3. Decentralized-control client-server

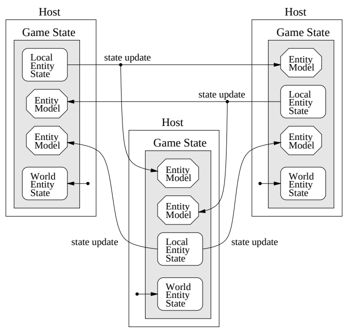

Fig. 4. Distributed

resolved by computing the resulting state.

We consider simulators where the progression of in-game time, called a *frame*, is distinct from *wallclock time*, and the simulator computes game state for each frame. Players take exactly one *turn* during each frame. Such simulators offer precise game control, but may appear slow to a player if the simulator cannot compute each successive unit of in-game time 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.

Multiplayer games are coordinated through either centralized client-server or decentralized distributed architectures. The architectures differ by where game state is maintained, which involves effecting changes and coordinating interactions. Under a *client-server* architecture, all entity states are maintained by the server, which computes game state based on input from clients and informs clients of the current game state. In a *centralized-control client-server* architecture, illustrated in Figure 2, a client sends requests for the server to change the client's entity state. In a *decentralized-control client-server* architecture, illustrated in Figure 3, a client informs the server of update decisions that have affected the client's entity state, and the server resolves interactions between game objects and coordinates global game state. Under a *distributed* architecture, where clients are referred to as *hosts*, each host maintains its own entity state, informs other hosts of decisions, and resolves any interactions without the use of any centralized authority. A distributed architecture, illustrated in Figure 4, is serverless and is also refereed to *peer-to-peer* gaming.

Client-server architectures offer a single point of game coordination at the server. In the decentralized-control client-server architecture, the server maintains an *entity model* that represents the last known entity state as updated by a client. In the face of missing 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, since the server has authority over maintaining game state, the client must accept these discrepancies and conform to the server's view of the game world. These simplifications, among other benefits, have made the client-server architecture popular. An entity model is also used by distributed hosts for remote entities. However, interaction decisions cannot be made in the face of missing entity state without potentially corrupting the overall game state. We explore this further in the next section.

Each architecture presented results in various potentials for cheating. *Cheating* occurs when a player causes updates to game state that defy the rules of the game or gains an unfair advantage. For example, players may cheat by using some piece of game state that a player is not permitted to have knowledge of according to game rules.

A. Related Work

Dead reckoning is a technique that compensates for variable communication latency and loss across a network by allowing a host 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 [7], [8], and is commonly used by researchers and developers [9], [10], [11], [12], [13], [14], [15]. 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 have refined this basic formulation [13]. Diot, Gautier, and Kurose have evaluated the performance of *bucket synchronization* with dead reckoning in a simple, distributed game called MiMaze [11], [12]. Bucket synchronization provisions a series of buckets at each host, 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 (i.e., the game time has reached that bucket's assigned time unit), any missing updates are dead reckoned. That is, the absent player's entity model is guessed using the player's state from the previous bucket.

Previous work places dead reckoning as a necessary technology for timely game play but largely leaves the resulting problem of entity interaction resolution as a future enhancement. In this paper, we show that employing a dead reckoning scheme and resolving entity interactions are mutually exclusive. Also not addressed in previous work is the potential for cheating under synchronization schemes. We show that cheating is possible in a dead reckoning system. We provide solutions to the cheating problem in later sections of this paper. The next section shows that any form of dead reckoning leads to irresolvable player interactions, especially with a distributed architecture.

Our work is also related to interest management techniques for massively multiplayer games and applications [16], [17], [18], [19], [20], [21]. Typically, interest grouping is done on the basis of (x,y) grid-coordinates, a natural interest clustering for the application area of virtual environments. Section VI discusses how our work can leverage these techniques.

Lastly, our work is related to parallel simulation techniques [22], [23], [8]. 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 human participant to restart to previous points in the game. In this paper we allow for optimistic processing without the need for rollback.

III. Fair Playout

We define an online game as *fair* if state as perceived by every player is consistent with every other player's expectations, including the server, as defined by game rules. Fair resolution of game events can be complicated by the use of dead reckoning in several ways.

A. Irresolvable Interactions

As stated, a player cheats by causing updates to game state that defy rules of the game or gains 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 occurs as would be expected by games rules and taken action. The goal of this paper is to achieve cheat-proof, correct, and fair gameplay. The solutions must be scalable to many players in terms of signaling costs, and maintain playout such that delays in the advance of in-game time is minimally perceivable to players.

Dead reckoning can be used in client-server or distributed environments [11], [12] and the operation is the same. In the centralized case, interactions are resolved by the server uniformly, but unfairly. That is, the server is the sole authority over game state that is used to resolve interactions, so all clients are updated with the same resulting view of the world. However, 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. Due to the unreliability and latency variability of the Internet, this client-server unfairness is unfavorably tolerated by game players.

The annoyance of unfair decisions becomes damaging in a distributed architecture, as interactions based on dead-reckoned state may corrupt the global game state as seen by each host. Since dead-reckoned state may not be consistent between hosts, using dead-reckoned state to resolve interactions results in an unfair decision, according to other players. In the distributed context, fairness includes *correctness*. The result of resolving an interaction based on dead-reckoned state in a distributed architecture is the potential for incorrectness of overall game state. For example, say players B and C dead reckon player A to take no actions, when in reality A destroyed B (*reality* is defined as the game state when interactions are resolved fairly and correctly). If player B then destroys player C before A's actions are resolved, then game play is corrupted. Game time could be restored to a state before the interaction, but this is clearly not a fair or practical solution, although it is exactly the approach taken by the High Level Architecture's optimistic time management service [8]. 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 host.

B. Cheating Under Dead Reckoning

The possibility of cheating is widely disregarded in multiplayer games. Many games are designed around the client-server architecture, which provides some implicit security along with centralized control of game state. Distributed games are much more prone to cheating, but cheats are possible within a client-server game.

One security flaw under bucket synchronization is what we term the *suppress-correct cheat*, which allows a host 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 connection and is removed from the game (coordinating the removal of a player in a distributed game is beyond the scope of this paper). 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 nth 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 nth bucket, S sends a fabricated update that places S on the heels of A. As long as S sends plausible updates every nth bucket, A cannot confirm that S is cheating or not cheating; 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.

IV. Cheat-Proof Game Interaction

Most real-time strategy games require interaction resolution at each discrete unit of time, or turn, in the game. A stop-and-wait-type protocol similar to those used for reliable transport [24] can fulfill this requirement for client-server or 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 for all other players to decide and announce their turn for frame $t+1$, and receive the announcements of all other players, 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 by each host.

Such a scheme expects that each player will make a next-turn decision based on the current turn state, then send that decision to each other player. However, this represents another opportunity for cheating: a cheating player can simply wait until all other players have sent their 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 a cheating agent that sends the decision to raise shields 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, first we present a protocol where lookahead cheats and suppress-correct cheats are not possible. This protocol has performance drawbacks as all players' rates are restricted to that of the slowest player. In the second part of this section, we provide enhancements to allow fair play while allowing players to not wait for all players before advancing in game time.

A. Lockstep Protocol

To counter the lookahead cheat, we propose a stop-and-wait-type protocol with a decision commitment step. We call this secured version the *lockstep protocol*, which is sufficient for implementing *lockstep synchronization* as described below.

Suppose turn t is complete. Each player decides but does not announce its turn $t+1$. Each player instead announces a cryptographically secure one-way hash of its decision as a commitment, including randomized padding if necessary to avoid recognizable hashed decisions and avoid collisions [25]. Once all players have announced their commitments, players then reveal their decisions in plaintext. Hosts can easily verify revealed decisions by comparing hashes of plaintext to the previously sent committed value. Because each host 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 host is not required to commit its decision if all other hosts 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 slowest player. The reception of other player's packets is likely to be delayed by current network conditions. The next section presents 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.

A.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 [24]. 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.[1]

Theorem: The lockstep protocol is safe: no host ever receives the state of another host 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. The lockstep protocol is live: the frames each player resolves monotonically advance with wallclock time.

Proof: The safety of the lockstep property follows directly from the protocol specification. Let AN be equal to the current frame being resolved by an arbitrary player A. Initially, $AN = 0$. As per the protocol description, A announces a hashed version of the decision it has made for AN once it has received commitments from all other players for the same frame. A will not announce its committed decision for time $AN+1$ until decisions for time AN 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.

For liveness, let t_1 be the wallclock time at which arbitrary player A starts to resolve frame AN of the game. Let t_2 be the wallclock time at which all players learn the revealed decisions of all other players for frame AN; let $t_2 = \infty$ if this never occurs. Let t_3 be the wallclock time at which player A advances to time frame $AN+1$; 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 $AN(t)$ equal the value of variable AN at player A at wallclock time t. Let $AN(t_1) = i$. By definition of the protocol, we know $AN(t_2) = i$. 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,

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

t_2 is finite. Because the protocol is safe, we know that the value of AN is incremented to $i+1$ only at time t_3. It is clear from the statement of the algorithm that $AN(t)$ is a non-decreasing value over time. Because $AN(t)$ is non-decreasing, $t_2 < t_3$. 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 lockstep synchronization by decentralizing the game clock. Each host advances in time asynchronously from the other hosts, 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 host processing. We do not expect AS to be used to allow players with completely different network and host 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.

B.1 Spheres of Influence

Using AS, each player's host 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. Accordingly, a sphere of influence is the in-game area that encompasses all sources of potential influences on a player's upcoming decisions. 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.

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.

In AS-based games, each host may be making decisions for a different time frame and advancing its time frame independent of other hosts; details are described subsequently. Therefore, a

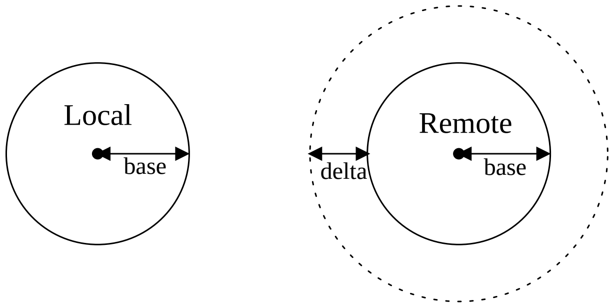

Fig. 5. Base and delta spheres of influence

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

Fig. 6. Table of variables.

```
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 $y \le t$
     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 $(t+1)$
```

Fig. 7. AS at the local player for each game turn.

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. 5, and delta is added to base to compute subsequent turns.[2]

B.2 Asynchronous Synchronization Protocol

Our description of the AS protocol is from the point of view of one host in a distributed, serverless architecture. The protocol can be easily adapted to a centralized architecture.

A formal description of the protocol is given in Figures 6 and 7 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 initial-

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

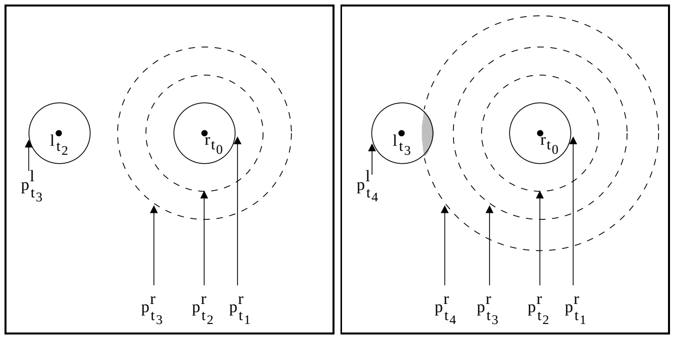

Fig. 8. (Left) Dilation to t_3. (Right) Dilation to and intersection at t_4.

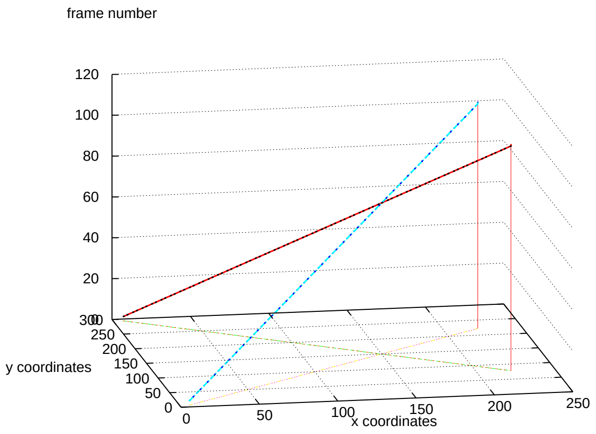

Fig. 9. Player position versus game frames. Top lines:players A, B, C, and D. Bottom lines: paths in the xy-plane.

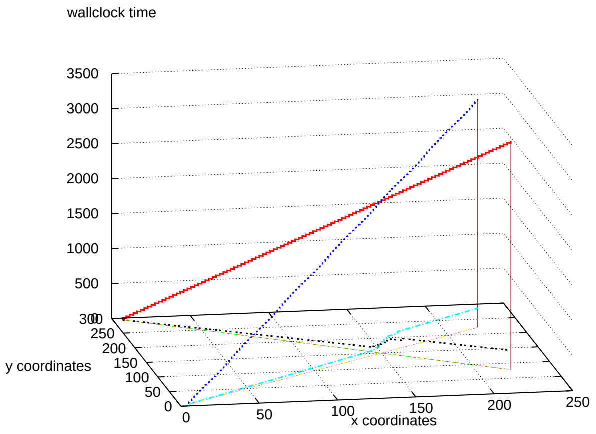

Fig. 10. Player position versus wallclock time. Top lines:players C and D. Middle lines: players A, and B. Bottom lines: paths in the xy-plane.

ization 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.

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 host 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. 8a). If the local host is not in the future of a remote host, 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. 8b.

Finally, if no remote hosts are in the wait state, the local host reveals its state for turn t, updates its local entity model of each other player with their last known state, including the remote host'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.

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

As an illustrative example, consider the case depicted in Figs 9 and 10, drawn from simulation results. Fig. 9 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. 10 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. 10 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 hosts 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 eliminates the suppress-correct cheat also in the same way as in the lockstep protocol. The host 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 host may purposely lag behind other hosts in order to preview future information. However, a host cannot advance in time past the point where a potential influence is detected, hence cheating is useless as no player would be affect by the cheat. (Note simple jumps in position are easily detectable as being located outside a dilated SOI from a known position). By definition of SOI, the future information released by the advancing host is immaterial to any other player's game decisions.

Note that 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. However, we present a solution to this situation for centralized and distributed architectures in Section VI.

The AS protocol also preserves lockstep synchronization's guaranteed correct and fair playout, since all interactions are resolved with perfect information for each turn in the game. The performance increase is the ability to advance in time independent of remote hosts when no interactions are possible. We demonstrate such performance gains by simulation in Section V.

A host 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 V make use of a similar type of capped rate.) At best, once a SOI intersection is detected, a host will have to wait for only one update from another player, which will restrict its potential SOI and allow the local host to continue. At worst, the potentially interacting player must catch up in time to a faster host 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.

B.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. 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.

C. *Secret Possessions*

Many games include the notion of secret information, which is valuable in-game state that may only be known to a subset of the players. One type is *secret possessions*, which are objects that a player chooses not to reveal immediately to other players, with the condition that no actions are taken with the object until it is revealed.

Players in a distributed game may wish to hide an object, such as a weapon or key, for a later use. However, opponents may desire proof that the object was in fact acquired earlier and within game rules. For example, in a capture-the-flag game, player A may capture an opponent's flag to bring back to a home base to win the game. Player A may hide which of its game soldiers has the flag causing the opponent to have to neutralize all of player A's soldiers running for home base when it is unclear which has the flag. To solve this dilemma, we define a *promise* as a place holder for secret possessions with the condition that the remote player claims no potential interaction exists with the hidden possession. Promises can be easily implemented in the form of a hashed or encrypted commitment sent to the opponent and later revealed. The opponent must sign the commitment to prevent repudiation that the secret was not promised. The promising player may choose to not continue on with the game until the promise is signed by the opponent.

To provide local player L the confidence that secretive, remote player S is acting honestly, we present a *cheat detection* scheme, which does not prevent cheating but allows it to be discovered. Along with the promise, S must commit to its current (secret) state, for example, using a one-way hash function as in the lockstep protocol. L records the committed state. The state is revealed and verified at a designated time, either after the game has been completed or based on some expiration after which old secret information is considered *declassified*, or no longer important to be kept secret. We define the *logger service* as a mechanism that resides in each host and records promised information to be *verified later*. To provide a more real-time cheat detection solution, we define the *observer service* as a trusted, centralized entity that receives secret information through secure channels and verifies it *during gameplay*. If cheating is detected, a protocol can alert players and prompt some action to be taken; e.g., removal from the current game and banishment from a tournament.

If L never knows the position of S, then asynchronous operation is not provided for L because L can only advance in time at the rate of S's promises, assuming S can only issue one promise per time frame. If L wishes to advance in time faster than S, then L must be certain that S's dilated, potential SOI does not intersect with its own, or potential interactions may be irresolvable. To preserve the benefit of asynchronous time advance, we provide a centralized solution. A *promise service* is a trusted, centralized entity that receives secret and non-secret information in the same way as the observer service. The promise service dilates S's SOI to L's time frame and issues a promise to L if no interaction exists. Assuming the promise service host can execute at least as fast as L's host, promises can be issued to L even in the case that S has not advanced in time. No secret information is revealed to L, and interaction resolution is guaranteed since the promise service in enacting the AS protocol. The promise service also solves the situation in which no player may know the entity state of another. SOI are dilated and promises issued on behalf of each player by the promise service. Introducing such a centralized solution countermines the potential for AS to be deployed for large, distributed simulated environments, and a distributed solution is left as future work.

V. 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 distributed architectures.

We are unaware of any work that presents analytical models of typical game play. Therefore, we took traces from a representative game, *XPilot* [26], 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.

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[3], 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 milliseconds 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 between players. 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 with a mean of 5 simulator time units. A simple exponential distribution was suitable for this preliminary investigation; our future work will include simulation on more varied distributions and network topologies. 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 non-playing 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 4 units of the simulation so that we were consistent with constraints on human reaction times assumed by previous simulation work [11], [12]; 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

[3]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.

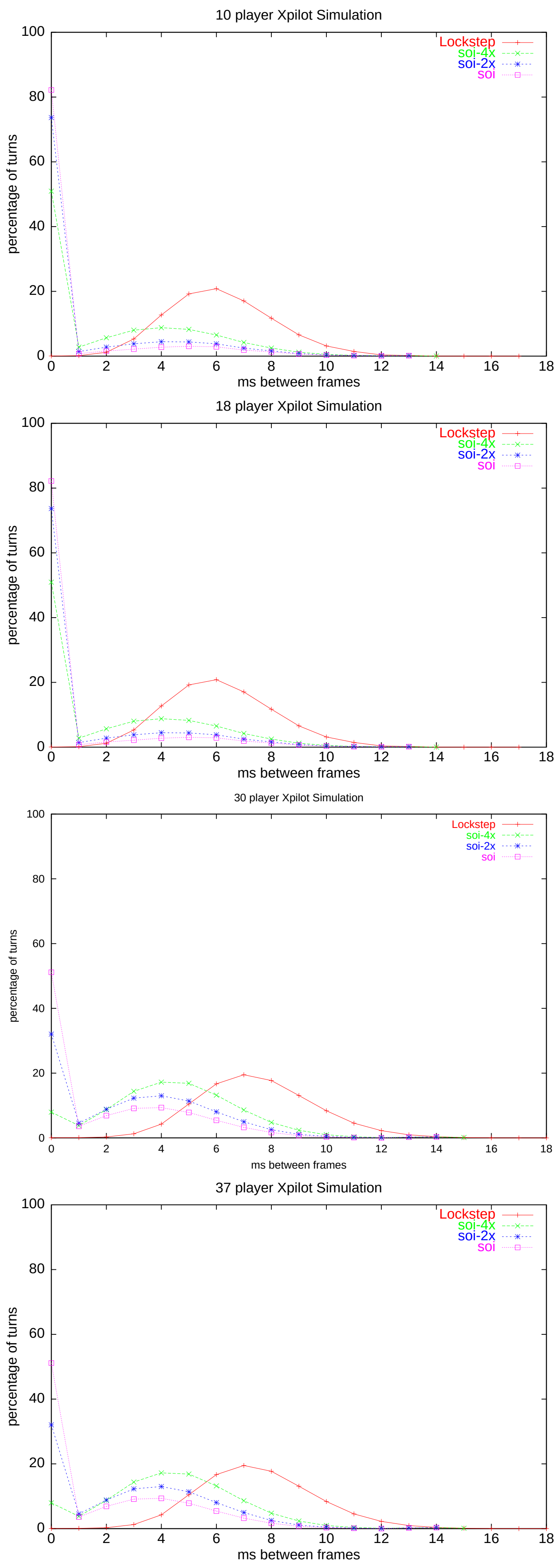

Fig. 11. Distribution of milliseconds stalled between frames due to lockstep interaction.

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, 18, 30, and 37 players. Our future work will include the study of larger number of players; XPilot is not a game suitable for play with players numbering in the hundreds or larger. 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 VI. 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. We expect this to be the size used in practice. The largest SOI simulated was of infinite size, corresponding exactly to a lockstep protocol. For comparison purposes, we also simulated twice the smallest SOI size (denoted soi-2x in the graphs), and four times the smallest SOI size (denoted soi-4x).

Fig. 11 shows the results for traces of 10, 18, 30, and 37 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 milliseconds 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.

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, even for larger numbers of players, at least 50% of the turns can be taken without delay do 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, so does the lockstep protocol, and that AS always maintains a large advantage. We expect these results to hold for large numbers of players in large environments, and our future work will be to test this theory. 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.

VI. Supporting Cell-based Architectures

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 geo-

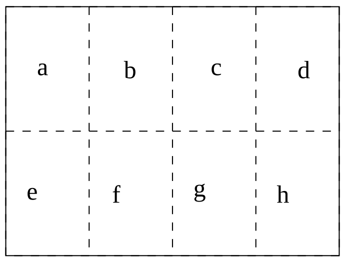

Fig. 12. A game arena divided into cells.

metric position. Accordingly, a virtual playing field may broken into *cells* to increase scalability [16], [17], [18], [19], [20], [21]. 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 III, 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 following technique allows players to discover whether they occupy the same simulation cell, without revealing to each other their current positions.[4]

A. Hidden Positions

Assume the game arena is divided into n cells, as in Fig. 12. Let each cell of the game be assigned a number from one to n. Assume player A is in cell $1 \leq x \leq n$ and player B is in cell $1 \leq y \leq n$.

A player would engage in this protocol whenever it decided to enter a new cell. Unfortunately, it must perform the exchange with all players to determine who is in the same cell. Optionally, once any player is found in the same cell, the in-cell player could inform the new participant of all other cell members. However, we do not discuss the details of such an optimization here.

Assume the cryptosystem used below is commutative (e.g., RSA [27] or Pohlig-Hellman [28]), so that for any message M we have $E_K(E'_K(M)) = E'_K(E_K(M))$, where E_K denotes encryption of a message with key K. This scheme also requires key exchanges between all players, which we do not specify

[4]We are indebted to Mikhail Atallah of Purdue University for suggesting the following technique for secure multiparty computation used in our solution.

here, but several methods exist already that may be employed (Schneier lists several [25]).

Step 1. Player A generates a random number R_a and sends player B a one-way hash of R_a; B generates a random number R_b and sends A a one-way hash of R_b. They now have both committed their choices to each other.

$$A \rightarrow B : h(R_a)$$
$$B \rightarrow A : h(R_b)$$

Step 2. They both compute R as the bitwise XOR of R_a and R_b. *Step 3.* A sends B the result of encrypting $(x + R)$ using a random key K generated by A and not known to B.

$$A : z = E_K(x + R)$$
$$A \rightarrow B : z, h(x)$$

Step 4. B sends A the result of encrypting $(y + R)$ using a random key K' generated by B and not known to A.

$$B : w = E'_K(y + R)$$
$$B \rightarrow A : w, h(y)$$

Step 5. A sends B the result of encrypting the w that she received in Step 3 using the key K that she used in Step 2.

$$A : w' = E_K(w) = E_K(E'_K(y + R))$$
$$A \rightarrow B : w'$$

Step 6. B sends A the result of encrypting the z that he received in Step 2 using the key K' that he used in Step 3.

$$B : z' = E'_K(z) = E'_K(E_K(x + R))$$
$$B \rightarrow A : z'$$

Step 7. A and B both learn whether $x = y$ by comparing z' to w' : $x = y$ if and only if $z' = w'$, which is true by the commutativity of the cryptosystem.

B could try to cheat by asking A to encrypt several choices. However, we require each player to later reveal their commitments from steps 4 and 6. In the worst case, this may be done at the end of a game.

VII. Conclusions

For the first time, 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.

Acknowledgements

We are grateful to Andy Fagg of the University of Massachusetts, Amherst and Mikhail Atallah of Purdue University for their helpful insight. The technology contained within this document has been protected worldwide by a patent.

References

[1] Blizzard Entertainment, “Starcraft,” http://www.blizzard.com.
[2] id Software, “Doom,” http://www.idsoftware.com.
[3] id Software, “Quake,” http://www.idsoftware.com.
[4] Heat.net, “10six,” http://www.10six.com.
[5] Turbine Entertainment Software Corporation, “Asheron’s call,” http://www.asheronscall.com.
[6] M. Pritchard, “How to hurt the hackers,” in *Game Developer Magazine*, pp. 28–30. June 2000.
[7] “Standard for information technology, protocols for distributed interactive simulation,” Tech. Rep. ANSI/IEEE Std 1278-1993, Institute of Electrical and Electronics Engineers, March 1993.
[8] F. Kuhl, R. Weatherly, and J. Dahmann, *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*, Prentice Hall PTR, Upper Saddle River, 2000.
[9] B. Blau, C. Hughes, M. Michael, and L. Curtis, “Networked virtual environments,” in *ACM SIGGRAPH, Symposium on 3D Interactive Graphics*, March 1992, pp. 157–160.
[10] E. Berglund and D. Cheriton, “Amaze: a multiplayer computer games,” 1985.
[11] C. Diot and L. Gautier, “A distributed architecture for multiplayer interactive applications on the internet,” in *IEEE Networks magazine*, vol. 13, pp. 6–15. Jul–Aug 1999.
[12] 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.
[13] 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.
[14] A. Watt and F. Policarpo, *3D Games: Real-time rendering and Software Technology*, Addison-Welsey, 2001, Ch. 20.
[15] J. Aronson, “Dead reckoning:latency hiding for networked games,” in *Gamasutra magazine*, September 19 1997, http://www.gamasutra.com/features/19970919/aronson_01.htm.
[16] D. Van Hook, S. Rak, and J. Calvin, “Approaches to relevance filtering,” in *Eleventh Workshop on Standards for the Interoperability of Distributed Simulations*, September 26-30 1994.
[17] M. Macedonia, M. Zyda, D. Pratt, D. Brutzman, and P. Barham, “Exploiting reality with multicast groups,” *IEEE Computer Graphics and Applications*, vol. 15, no. 5, September 1995.
[18] S. Rak and D. Van Hook, “Evaluation of grid-based relevance filtering for multicast group assignment,” in *Proc. of 14th DIS workshop*, March 1996.
[19] E. Léty and T. Turletti, “Issues in designing a communication architecture for large-scale virtual environments,” in *Proceedings of the 1st International Workshop on Networked Group Communication*, November 1999.
[20] K. L. Morse, *An Adaptive, Distributed Algorithm for Interest Management*, Ph.D. thesis, University of California, Irvine, 2000.
[21] B.N. Levine, J. Crowcroft, C. Diot, J.J. Garcia-Luna Aceves, and J. Kurose, “Consideration of Receiver Interest for IP Multicast Delivery,” in *In Proc. IEEE Infocom 2000*, March 2000.
[22] R.M. Fujimoto, *Parallel and Distributed Simulation Systems*, Wiley Interscience, January 2000.
[23] R.M. Fujimoto, “Time management in the high level architecture,” *Simulation*, vol. 71, no. 6, pp. 388–400, December 1998.
[24] D. Bertsekas and R. Gallager, *Data Networks*, Prentice-Hall, Englewood Cliffs, 1987.
[25] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons, Inc., New York, second edition, 1996.
[26] B. Stabell and K.R. Schouten, “The Story of XPilot,” ACM Crossroads Student Magazine, Winter 1996, XPilot software available from http://xxx.xpilot.org.
[27] L. Adleman, R. L. Rivest, and A. Shamir, “A method for obtaining digital signature and public-key cryptosystems,” *Communication of the ACM*, vol. 21, no. 2, 1978.
[28] S. C. Pohlig and M. E. Hellman, “An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance,” *IEEE Trans. Inform. Theory*, vol. IT-24, pp. 106–110, Jan. 1978.