

Supporting P2P Gaming When Players Have Heterogeneous Resources

Aaron St. John

Brian Neil Levine

Dept. of Computer Science, University of Massachusetts, Amherst, MA 01003
{stjohn,brian}@cs.umass.edu

Abstract

We present Ghost, a peer-to-peer game architecture that manages game consistency across a set of players with heterogeneous network resources. Ghost dynamically creates responsive sub-games based on the delay profiles of players. 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. Ghost extends our earlier Asynchronous Synchronization (AS) protocol, which provides cheat-free payout for peer-to-peer games. This modification to AS enables p2p games to efficiently function in network environments that would typically be hostile to multiplayer networked games. These include networks with highly variable delays and variable route partitions. Our evaluation shows that Ghost performs well, always ensuring consistent p2p play with the maximum number of players, while preventing any one player from destroying the quality of play for others.

1 Introduction

Like all networked, multi-party applications, managing a game between multiple users with heterogeneous connections is a challenge. Even if all users have comparable machines and connections to the Internet, connections between users may vary due to inter-ISP resources. For example, one clique of users may have high-speed connections to each other, but may be unable to maintain resourceful connections to another subset. In the worst case, one slow player can dictate the speed of the entire game for all players.

Client-server game architectures solve this problem by requiring clients to maintain sufficient bandwidth only with a server; thereby pushing the majority of the bandwidth burden onto a central machine (or cluster of machines.) Indeed, commercial game companies spend enormous resources to ensure that the servers they host are well provisioned to reach all parts of the Internet. For peer-to-peer games, such an approach is infeasible. We believe the best approach for both peer-to-peer and client-server games is to dynamically create cliques of players each have network resources sufficient to satisfy the constraints set forth by constituent players.

In this paper, we present *Ghost*, a peer-to-peer game architecture that manages game consistency across a set of players with heterogeneous network resources. Ghost dynamically creates responsive sub-games all in the same virtual world based on the delay profiles of players. Ghost allows each user to set the quality of game they are willing to tolerate, and then it manages game consistency while creating the maximum-sized game that satisfies the users' requirements. Ghost leverages a unique *connectionless* consistency model to improve performance: if player p is *playing with* player q , it is a non-reflexive relationship. It implies only that p is willing to wait for q 's upcoming game actions until a maximum delay is reached; it does not imply that q is playing with p . Because p and q attempt to maximize the number of peers they play with, they will effectively have a two-way relationship. However, because the relationships are one-way, Ghost has no hard-state agreements or reliable transfers that must be set up or torn down, delaying stalled game progress. Moreover, Ghost is able to enforce *team play*. In short, this means that if p and q are on the same team, then p will not play with r if r was not willing to wait for q earlier in the game. Determining these relationships does not stall the game for any player.

This work was supported by National Science Foundation awards CNS-0133055, CNS-0323597, and EIA-0080199. NOSSDAV, June 13–14, 2005, Stevenson, Washington, USA. 1-58113-987-X/05/0006.

Ghost extends our earlier Asynchronous Synchronization (AS) protocol, which provides cheat-free ployout for peer-to-peer games. This modification to AS enables p2p games to efficiently function in network environments that would typically be hostile to multiplayer networked games. These include networks with highly variable delays and variable route partitions. Our evaluation shows that Ghost performs well, always ensuring consistent p2p play with the maximum number of players, while preventing any one player from destroying the quality of play for others.

Ghost works only with *membership independent* games, which is the paradigm of most currently popular commercial games (e.g., Quake or Unreal Tournament). Such games are not disturbed by players joining and leaving the session without notice.

We evaluated the Ghost architecture in a simulation that modeled Internet-like delays. In all cases, we found that Ghost performs as we desired: all users play games without consistency problems and never wait longer for consistency resolution than their pre-set limits.

2 Background

Much recent work on games and delay has focused on the impact of delay on playability [10, 3, 8]. New techniques for reducing the impact of delay and lag on the multiplayer game experience has received less attention.

One approach to managing delay is to spend resources. Commercial client-server games have been based largely on high-quality connections to a resourceful server. This is the simplest and most efficient method for providing multiplayer games. However, servers can be costly and troublesome to maintain, and they act as a single-point-of-failure for any game.

Another approach is to design better algorithms. *Dead reckoning* [1] is the traditional protocol approach for mitigating network delays between clients and servers. However, it can compensate for only small amounts of delay and jitter in the network. Diot et al [5, 7] proposed using dead reckoning for p2p games. However, our past work has shown [2] dead reckoning is vulnerable to cheating and can cause inconsistencies in game ployout.

Peer-to-peer games distribute the load across the peers playing the game at the cost of requiring resourceful pairwise connections between peers. In a

naive p2p game architecture, the peers all have pairwise connections to each other and move in lockstep. During each frame of play, the peers must broadcast their move to the other players in the game. Once all moves have been heard by all peers, they all advance to the next game frame. This ensures consistency, but also advances at the rate of the slowest player. Ideally, peer-to-peer games would advance despite the slowest player, which is what we propose in this paper.

Many Massively Multiplayer Games use in-games zones as a means to distribute load across several servers. These zones are typically static features of the game world which require a full disconnect/reconnect cycle to travel between, causing a disruption in game play. Our technique seeks to distribute load dynamically across proximal peers without interrupting the flow of game-play more than a pre-specified amount of tolerable disruption.

In our own previous work, we proposed the Asynchronous Synchronization (AS) protocol for peer-to-peer games. AS maintains the consistency achieved by a lockstep protocol but avoids the overhead of forcing all players to resolve moves at each frame. Thus, AS can advance game play in the presence of jitter or short-term temporary game stoppages of one player. Since Ghost is an extension of AS, we review the important details of AS below.

In addition to our own work on the AS protocol [2], at least a few other papers have appeared on cheat-free serverless distributed games. Those protocols, by Cronin et al [4] and Lee et al [9], are extensions to AS. They extend the stop-and-wait lockstep protocol for cheat-proof ployout to provide a go-back- n , pipelined approach to coordinating player moves. However, those approaches, like AS, do not prevent resource-poor players from degrading the game performance of other peers — in this paper, we solve that problem. A voting-based solution has been proposed by GauthierDickey et al [6] where late packets are accepted only if a majority of other players have received them.

2.1 Asynchronous Synchronization

AS maintain p2p game consistency of a loosely synchronized game *frame* clock. Players track a *sphere of influence* (SOI) around each remote peer. The area covered by the SOI expands to include all possible moves made by the remote peer since the last update was received. As a local player's frame clock advances, she dilates the spheres of peers that have

not sent updates. Dilation of the sphere is analogous to how a buffer is used in a multimedia stream; it uses the game topology and physics as a buffer for delaying consistency resolution. After a lengthy delay, a remote peer’s SOI intersects with where the local player can move to in the next frame (represented by a unit-radius sphere around the local player). At this point, a type of *lockstep* protocol begins, involving only the intersecting remote peer. Lockstep peers commit to their next updates with hashes (e.g., SHA-1 or MD5), and then reveal. This process prevents certain types of cheating detailed in the original paper [2].

The key performance benefit of AS is that players need not interact until their SOI intersect. Thus, if a player’s updates are slow, they only affect a small number of other players if network performance problems are short term. Peers can resolve game state per frame with only partial game information.

However, the AS protocol cannot handle long-term delays or game stoppages: the game is played at the speed of the least-resourceful player. In fact, any one player can stop the game entirely until other players timeout and remove them from the game. Ghost addresses these problems, extending AS to account for heterogeneous delays among players.

3 Ghost

Ghost is based on several key ideas regarding delay, consistency, and connectionless relationships between peers. This section details the operation of Ghost among players on the network. Ghost operates as an extension to AS; in practice, this would be a single protocol, but for clarity, we maintain their separation in the description below.

3.1 Overview

In Ghost, each user sets a required minimum rate for advancing the game frame clock, which we call the *threshold frame rate*, denoted as f . Each player manages consistency with other peers using the AS protocol. However, users will not play with remote peers that prevent them from advancing the game at a rate that is slower.

At a local player, p , Ghost divides the group of remote peers into two subsets: an *active set* and an *inactive set*.

For each member, q , of the active set, the local peer follows the AS protocol, maintaining consistency with

q by delaying the frame clock until q ’s moves are determined not to conflict (by calculating that the SOI do not intersect) or by entering lock step (if the SOI do intersect). We say that p *plays-with* q if p is willing to wait for lockstep resolution of q ’s actions. It is important to realize that playing in lockstep is equivalent to playing the game. By definition, it is only at such close proximity that can players affect each other in the game through their actions. If, for example, a player disconnected from the network temporarily just before they perished in the game, then that violates the plays-with rule. This is an critical issue, which we address in Section 3.5.

For each member of the inactive set, r , the local player does not follow the AS protocol and will reveal moves regardless of the state of r . (Hence our protocol name: inactive peers are like *ghosts* that inhabit the same world but have no influence.) Note that inactive members continue to broadcast at least simple ping messages periodically so that others can determine current network conditions.

Active peers are those that meet the threshold frame rate requirements. Although the underlying mechanisms in AS allow some buffering of short-term network jitter. The active peer is *booted* if jitter becomes a long-term problem, becoming an inactive player from the perspective of the booting peer.

The concept in Ghost that is most unintuitive is that its *play-with* relationship is not reflexive. If p plays-with q , it does not imply that q plays-with p . The reason this is possible is that, unlike the AS protocol, with Ghost, p will not delay endlessly while waiting for q ’s move. Ghost can maintain a non-reflexive plays-with relationships that does not require agreement, notification, and does incur an unnecessary delay.

Ghost distinguishes between delays that are due to the consistency requirements of lockstep exchanges, delays that are due to network jitter, and long-term delays that seriously threaten game ployout quality. If a player experiences only short-term network delays in exchanging updates with others, AS will smooth ployout in the short term by dilating the remote player’s sphere of influence. If a player, p , experiences long-term delays from a remote player q , then the SOI around q will intersect with the SOI of player p . In the original AS protocol, p would simply wait indefinitely for q ’s update; even worse, p ’s delay may cause a chain reaction of delays at other players. However, in Ghost, player p checks the *delay profile* it has measured of q ’s performance in the game over the last n

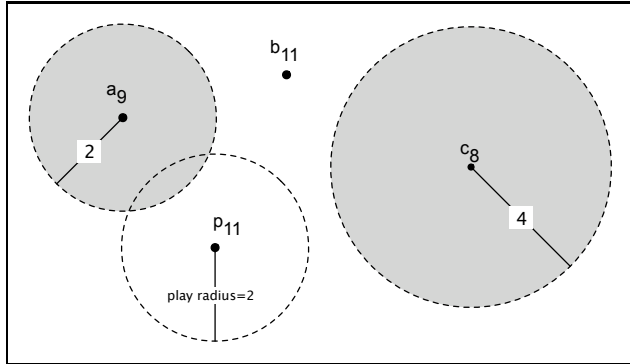


Figure 1: Player p joins by setting its clock to the maximum frame clock held at remote peers. All players with SOI dilated spheres outside the play radius are played-with.

rounds. If the average delay plus the standard deviation of the delay of receiving updates from q is below p 's threshold frame rate, then q is removed from p 's game. We refer to removal as *booting*.

Another key aspect of Ghost is that booted players and late joining players continuously attempt to connect and play with any remote peer whose delay profile is sufficiently above threshold. We refer to this process as *respawning* and it occurs seamlessly during game play without delaying the game.

Unlike existing games, Ghost's rules result in an odd circumstance: users may be each interactive with a different set of remote players. In Section 3.5, we shown how to enforce team play in Ghost. If p and q have a *team agreement*, then p will not play with any player that cannot meet q 's frame rate requirement.

In this section, we detail how these key ideas work together in Ghost. Due to space limitations, we have omitted some lesser details. For example, we assume there is a bootstrapping peer from which users can learn the IP addresses of remote peers. Ghost requires either application-level multicast, network multicast, or pairwise connections among peers. We assume that players receive updates without joining a new multicast group or equivalent structure. Ghost is easily emended to include the requirement of explicit joins. We assume peers periodically refresh their current state so that explicit retransmissions are not required.

3.2 Initialization

To join the game, the new user, p , learns the list of remote peers from the bootstrap server. Then, p

contacts other peers to learn their current location in the game topology and current frame clock value. It then picks a random location in the game topology that is not proximal to other peers. Next, it sets its frame clock to the maximum value of all frame clocks of the remote peers. Then, p dilates the spheres of all other players to a radius equal to the difference with its new frame clock. It also sets a *boot radius* around its current location. The boot radius accounts for network delay that can occur between peers and, as we explain below, adds smoothness to the payout. Finally, p sets as its active list all peers that who have dilated SOI that do not intersect with the boot radius.

This final step is illustrated in Figure 1. Player p has determined that the maximum frame clock is at player b . It then dilates player a by 2 units and player c by 4 units. Player p will then place b and c in its active set. As we detail below, player p will try to include player a in its active set later.

During broadcasts of its state, a player includes a list of players in its active set.

3.3 Active Set Interactions

After initialization, players engage in the AS protocol with all peers in their active set. We discuss the inactive set subsequently. During play with the active set, the local player maintains the same *boot radius* set in the initialization phase. The boot radius acts like a warning line: players with SOI that dilate past the boot radius might be booted. The goal is to boot players before they cause unnecessary delays.

Unlike the AS protocol, Ghost distinguishes two different causes of delays in advancing each player's frame counter and handles them differently. Both types are illustrated in Figure 2.

First, the proximity of two players will force them into a lockstep exchange of moves. If two players' next movements place them within one sphere of influence of each other, then they must first exchange hash commitments of their updates, reveal their moves, and then advance their frames. This type of delay, which we call *proximity delays* cannot be avoided, and it does not represent a long-term problem with insufficient network connections. In Figure 2, player p experiences this type of delay with player r .

Second, if updates are delayed between two players that were in remote positions at their last exchange of moves, their SOI will dilated and intersect. This delay ends when an update is received for frame f or

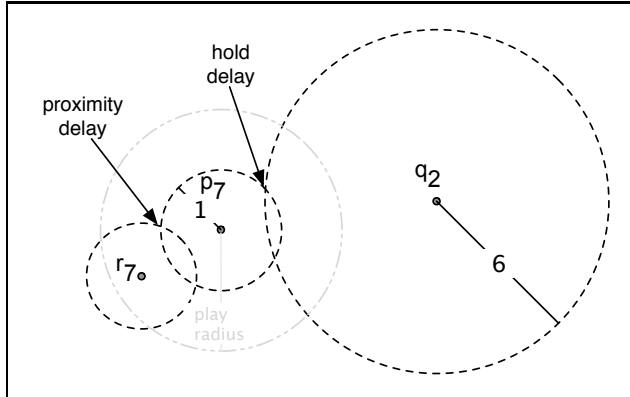


Figure 2: Three peers p , q , and r in a p2p game, from p 's perspective at frame 7. The last confirmed update p received from q was from game frame 2; the last update from player r was the same frame, but they are too close to avoid lockstep interaction.

some previously unreceived earlier frame. We refer to this type of forced wait as *hold delays*. In Figure 2, player p experiences this type of delay with player q , who has not sent an update since frame 2.

The goal of Ghost is to prevent hold delays before they occur using the boot radius. When a remote peer fails to send updates, eventually the dilated SOI will cross the boot radius. At this point, before hold delays occur, the local player will boot the remote peer based on a history of recent round trip times, which we call the *delay profile*. If the profile shows that delays are more often above the threshold frame rate, the player is booted; if not, the SOI continues into lockstep. We detail this exact process below.

Ghost players also timeout while waiting for resolution of proximity delays if necessary; however, the goal is to avoid reaching a close proximity to any player with a large delay profile.

3.3.1 Delay Profiles and Booting

To create delay profiles of other players, each player keeps track of the last n round-trip times between itself and all other peers. In our implementation we set $n = 50$. A player may be actively playing with only a subset of peers, but measurements to all peers listed by the bootstrap server are maintained¹.

To perform the measurements, all messages broadcast from the local peer include a piggybacked local

¹Some finite limit of peers exists, clearly. Additionally, peers may stop tracking peers that have chronically poor round-trip delays.

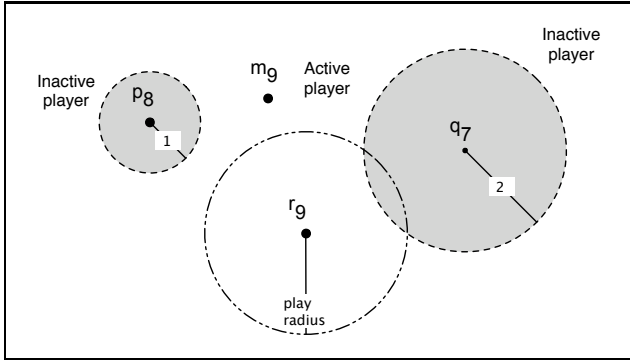


Figure 3: Respawning in Ghost. Player r can respawn with player p but not player q , who borders into the boot radius. Player m is already an active player.

timestamp. When exchanging updates with a remote peer, the timestamp of the latest message is returned with the accumulated processing and queuing delays.

The window of n_p data points for remote player p is summarized as a mean, $\overline{n_p}$, and standard deviation, σ_p , in milliseconds. The threshold frame rate for a peer is represented as a number of milliseconds, and represents the delay allowed between frame advancements. A remote player is booted if three conditions hold: the SOI of the remote player is within the boot radius; the delay is not due to proximity; and $\overline{n_p} + \sigma_p > f$. For example, if delays are normally distributed, players are booted if more than about 16% of their delays are below threshold. It should be obvious that any metric can be substituted to enforce different performance standards.

3.4 Respawning the Inactive Set

Ghost attempts to make members of the inactive set part of the active set, a process we call *respawning*. It follows the same pattern as the initialization process. Respawning occurs during game play and without interruption or delay.

To respawn, the local player creates a *candidate* set of players in the inactive set that have a delay profile that is above threshold. The local player checks the latest update of each candidate for the latest location and frame clock value (joining the appropriate application level multicast group if necessary). The local player determines the maximum frame clock of all players in the active set and of all candidates. It then dilates the SOI of all candidate players to this frame. The local player then moves any candidates to the active set if these dilated spheres fall outside

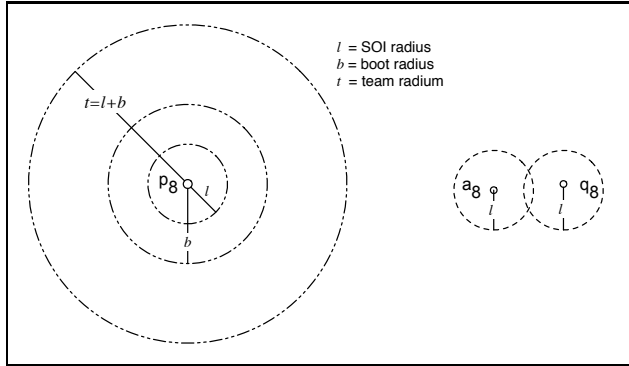


Figure 4: Enforcing team play.

the boot radius.

Figure 3 illustrates this process for a node r joining the game at frame 11. In the figure, player m is already in the active set and player p and q are inactive but have sufficient delay profiles. Player m has the maximum frame clock. Player r is able to play-with player p , but not player q , whose dilated sphere falls within the boot radius.

The interesting aspect of this process is that it happens during game play. Players can spawn every frame when they don't intersect with any other player, and can be spawned to every frame if they don't intersect with the remote player.

3.5 Team Play

Many games are based on team play and this is easily supported in Ghost. Given our play-with relationship, team play in Ghost means that if p is on a team with q , then if a is playing-with q , it must also play-with p . More specifically, team play means that if a stops playing-with q during lockstep then Ghost must prevent p from playing-with a . Therefore, our goal is that p must receive an update from all teammates who could have been in lockstep with a before a enters the boot radius of p .

To enforce team play, we introduce a third sphere or radius t around each player called the *team radius*. We define it to be twice the size of the SOI radius (which is always equal to $l = 1$) plus the boot radius (which is $b = 2$ in our implementation). Therefore, $t = l + b$. Teammates and non-teammates are handled differently now.

Whenever the SOI of a remote teammate intersects with the team radius (due to proximity or dilation from late updates), the local player requires an update of who the remote player is playing-with. Recall

that a list of each player's active set is piggybacked on all broadcast messages. If the update is not received and if a non-team player's SOI is within the team radius, the frame clock will stop, delaying play. It is important to note that a remote teammate can broadcast its active set list even if it cannot advance its frame clock. Note that team members cannot be booted at all once they are inside the team radius. This is the cost of having teams. A player can leave a team if all teammates are outside its team radius.

Given the sizes of the spheres, whenever the SOI of a non-teammate intersects with the boot radius the local player is guaranteed to have already been updated by all team members that the opponent may have broken lockstep with. This is because any teammate that could have been in lockstep with a had to have been within $l = 1$ radius of a 's position at some point in the past. This is illustrated in Figure 4. It is easy to show that the distance from p 's sphere to q 's sphere minus t is always less than p 's sphere to a 's sphere minus b if q was in lockstep with a . Informally, this is because $\overline{PQ} < \overline{PA} - 1$ and therefore $\overline{PQ} - 1 - t < (\overline{PA} - 1) - 1 - (l + b) < \overline{PA} - 1 - b$. That is, q 's update of its active set will always arrive before a 's sphere enters p 's boot radius. Note that if $t = b + b$ then opponents cannot boot team members from their boot radius without consequence.

While nothing can stop any opponent from booting any player, we suggest that team play provides an incentive for users to not disrupt play just before they perish in the game; if they do, whole teams will not play with them.

3.6 Limitations and Extensions

There are some simple extensions to the Ghost protocol that allow it to be applicable to a wider variety of games.

For games that are not tolerant to peers frequently dropping in and out, we can ensure that opposing players always play together during the duration of a game. In a slight twist on Ghost's team play feature, we can place opposing players on same "team". This technique uses Ghost's adaptive approach to delay to allow players to accept any amount of delay from their opponent in order to maintain the consistent membership requirement.

Since Ghost's packet size grows linearly with the number of peers participating, there is a limit to the scale of games driven by Ghost. As a simple method for scaling to a larger number of peers, Ghost can take advantage of cell based techniques discussed in

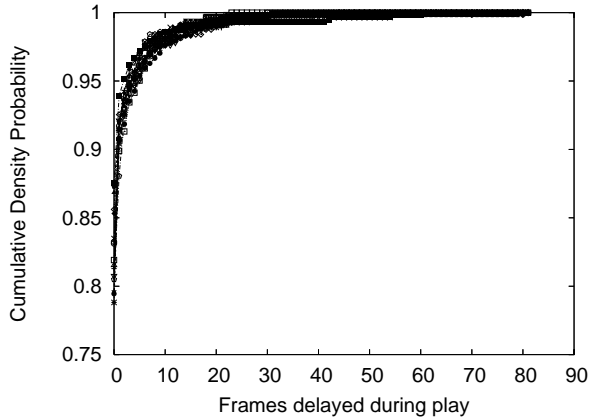


Figure 5: CDF of hold delays at peers in AS.

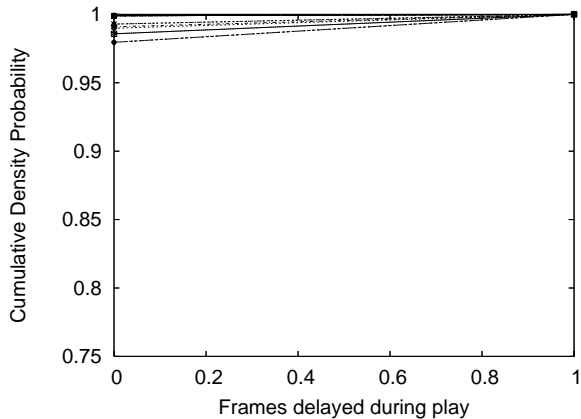


Figure 6: CDF of hold delays at peers in Ghost.

[2], which is a well known technique. Using cells, peers need track their membership only with peers who occupy the same cell. Peers would simply modify their membership lists as remote peers enter and leave cells.

4 Evaluation

Ghost seeks to dynamically partition a network game into the smallest set of sub-games in which peers can play with each other responsively within some user-defined delay tolerance. In this section, we show by simulation two aspects of Ghost. First, that in our simulations it prevented all peers from waiting longer than the maximum amount of delay defined as tolerable by that peer. Second, that in the simulations the size of the active sets of each peer was maximum number for that amount of tolerable delay.

Our simulation models the behavior of a simple peer-to-peer game managed by Ghost. Each peer is represented by a point and direction vector within a rectangular 2D world. Each frame a peer modifies their position by their direction vector and speed of travel. A peer’s direction vector is changed on impact with the boundary of the world. The speed at which each peer moves per frame is constant. In this model, one sphere of influence can be represented by a circle around the peers current position with radius equal to the peer’s speed. The simulation is written in Java and performs Ghost over a simulated network in which the delay between peers and rate of packet loss can be determined from various distributions, as we detail below.

4.1 Results

First, we verified that for each peer under Ghost, the amount of per frame delay experienced by that peer during the simulation was always less than that peer’s specified delay tolerance. We examined the amount of time a peer had to wait to transition from one frame to the next as a result of playing with another player. We ran the Ghost simulator for 10,000 simulation frames on 16 peers with pairwise delays taken from a random exponential distribution. The mean for each peer was chosen randomly from a uniform distribution of 1 to 30.

Figure 5 shows the performance of the players when only the AS protocol is used. The figure shows the CDF of hold delays for all peers (the plots are stacked on one another). Hold delays account for 20% of game time for all 16 peers with AS alone. Figure 6 shows the same simulation when Ghost is used. The CDF is almost meaningless because peers do not wait for other peers except during warm-up when the delay profiles are being initialized. Each peer was given the same tolerance for delay. The simulation results show that no peer waited longer than their delay tolerance for any other peer.

Our second evaluation was on how Ghost groups peers. Our grouping analysis shows that Ghost causes the largest possible responsive sub-games to be formed. In this simulation we distributed delays such that subsets of peers had good connections to each other with 10ms delays. For peers outside the subsets, connections had a 50ms delay. Both delays were chosen from an exponential distribution. Delay tolerance for all peers was 25ms. Again, we ran the simulation for 10,000 frames on 16 peers. We then analyzed the frequency at which peers played-

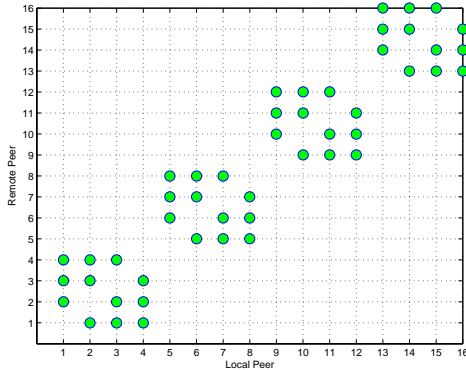


Figure 7: Which peers played-with one another.

with each other when in play radius. We found that Ghost ensured that peers whose pairwise delay fell below the tolerance always played with one another, this had the effect of breaking the game into four distinct responsive sub-games. This is illustrated in Figure 7, which shows a table of which peers played-with one another. A circle indicates the peer listed on the x-axis always played with the peer listed on the y-axis. The symmetry of the table demonstrates that the groups were consistent.

5 Conclusion

Network games have not received the attention or comprehensive discussion that has been paid to networked audio, video, or other multimedia systems. Yet gaming can be the most challenging application of all to support. Massively multiplayer, online games present consistency challenges coupled with adversarial peers while requiring updates generated at multimedia speeds from multiple, remote sources.

We have proposed Ghost as a method to support peer-to-peer games when players have heterogeneous resources. We have included details on joining, booting, and respawning of players that do not introduce delay in the game for peers. Additionally, we have shown how team play is supported in Ghost, which also acts as an incentive for peers to not boot opponents whenever they are about to perish. Our protocol uses a unique connectionless plays-with relationship, which allows Ghost to avoid delays due to hard-state relationships.

References

- [1] J. Aronson. Dead reckoning: latency hiding for networked games. In *Gamasutra magazine*, September 19 1997. http://www.gamasutra.com/features/19970919/aronson_01.htm.
- [2] N. E. Baughman, M. Liberatore, and B.N. Levine. Cheat-Proof Payout for Centralized and Severless Online Games. *IEEE/ACM Transactions on Networking*. To appear.
- [3] T. Beigbeder et al. The effects of loss and latency on user performance in unreal tournament 2003. In *Proc. ACM NetGames*, pages 144–151, 2004.
- [4] E. Cronin, B. Filstrup, and S. Jamin. Cheat-proofing dead reckoned multiplayer games. In *Proc. Application and Development of Computer Games*, 2003.
- [5] C. Diot and L. Gautier. A distributed architecture for multiplayer interactive applications on the internet. In *IEEE Networks*, volume 13, pages 6–15. Jul–Aug 1999.
- [6] C. GauthierDickey, D. Zappala, V. Lo, and J. Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *Proc. NOSSDAV*, pages 134–139, 2004.
- [7] L. Gautier, C. Diot, and J. Kurose. End-to-end transmission control mechanisms for multiparty interactive applications on the internet. In *Proc. INFOCOM*, 1999.
- [8] T. Jehaes et al. Access network delay in networked games. In *Proc. ACM Netgames 2003*, pages 63–71, 2003.
- [9] H. Lee, E. Kozlowski, S. Lenker, and S. Jamin. Synchronization and cheat-proofing protocol for real-time mulitplayer games. In *In Proc. of Intl Workshop on Entertainment Computing*, May 2002.
- [10] N. Sheldon et al. The effect of latency on user performance in warcraft iii. In *Proc. ACM NetGames 2003*, 2003.