# Bandwidth requirement and state consistency in three multiplayer game architectures

Joseph D. Pellegrino
Department of Computer Science
University of Delaware
Newark, Delaware 19711
Email: jdp@elvis.rowan.edu

Constantinos Dovrolis
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332
Email: dovrolis@cc.gatech.edu

*Abstract*— **Multiplayer games become increasingly popular, mostly because they involve interaction among humans. Typically, multiplayer games are organized based on a Client-Server (CS) or a Peer-to-Peer (PP) architecture. In CS, players exchange periodic updates through a central server that is also responsible for resolving any state inconsistencies. In PP, each player communicates with every other player while state inconsistencies are resolved through a distributed agreement protocol.**

**In this paper, we first examine these architectures from two perspectives: bandwidth requirement at the server and players, and latency to resolve any player state inconsistencies. Our results are based on both analysis and experimentation with an open-source game called "BZFlag". The CS architecture is not scalable with the number of players due to a large bandwidth requirement at the server. The PP architecture, on the other hand, introduces significant overhead for the players, as each player needs to check the consistency between its local state and the state of all other players. We then propose an architecture that combines the merits of CS and PP. In that architecture, called Peer-to-Peer with Central Arbiter (PP-CA), players exchange updates in a peer-to-peer manner but without performing consistency checks. The consistency of the game is checked by a central arbiter that receives all updates, but contacts players only when an inconsistency is detected. As a result, the central arbiter has a lower bandwidth requirement than the server of a CS architecture.**

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
NetGames2003, May 22-23, 2003, Redwood City, California.
Copyright 2003 ACM 1-58113-734-6/03/0005 $5.00.

## I. Introduction

The recent popularity of first-person shooting multiplayer games brings along the complexities of bandwidth management and state consistency [1]. Bandwidth requirement is directly related to the scalability of a game. If, say, the bandwidth requirement at a game server increases quadratically with the number of players, the network connection of that server will probably become a bottleneck as the number of players increases. Another important issue about multiplayer games is that they need to maintain state consistency among players [2]. For instance, in racing games all players should have the same view about the position of each car, while in first-person shooting games all players should agree on who is alive and who is dead.

In this paper, we first examine two major architectures for multiplayer games. These architectures are based on the *Client-Server* (CS), and the *Peer-to-Peer* (PP) models. We analyze these models from two perspectives: bandwidth requirement and inconsistency resolution latency. Then, we propose a new architecture called *Peer-to-Peer with Central Arbiter (PP-CA)*. With the CS architecture, it is simpler to maintain state consistency, but the server bandwidth requirement increases quadratically with the number of players. With the PP architecture, the server bandwidth bottleneck is removed, but resolving state inconsistencies requires a distributed agreement protocol [3]. With the PP-CA architecture, a centralized arbiter resolves any state inconsistencies, but with a lower bandwidth requirement than the server of a CS architecture. The reason is that the central arbiter of the PP-CA architecture contacts the players only when it detects an inconsistency, while the server of the CS architecture contacts the players in each update period. We use an open-source "capture-the-flag" game

called *BZFlag* to experimentally measure the bandwidth requirement and consistency resolution latency of the previous three architectures.

### A. Bandwidth Requirement

Players often use limited-bandwidth links, such as dialup and DSL modems, and so the bandwidth requirement of a game is of major importance to them. In a CS architecture, the server has often (but not always) a network access link of higher bandwidth than the players. Even when this is the case, however, we cannot ignore the bandwidth requirement at the server. If the server bandwidth requirement increases rapidly with the number of players, its access link can become the game's bottleneck.

In a typical multiplayer game, a player executes a certain loop throughout the duration of the game. This loop includes reading the input of the local player, receiving updates from remote players, computing the new local state, and rendering the graphical view of the player's world. The duration of this loop is referred to as the *player update period* $T_U$. While $T_U$ can vary among different computers playing the same game, it is typically fairly stable at the same machine. In this paper, we will assume for simplicity that all players have the same update period. Each player sends an update message in every iteration of the previous loop to either all other players, or to a central server. The size of the update messages can vary, depending on the activity of the player during that loop iteration. To simplify the bandwidth requirement calculations, we will also assume that all updates have the same size, namely $L_U$ bytes.

So, the bandwidth requirement of a player or of the server are determined by the number of update messages sent and received in each update period $T_U$. This number of messages depends on the underlying game architecture, and on the way state inconsistencies are detected and resolved. In the following, when referring to the bandwidth requirement at a certain node, we use the term *upstream* to refer to data sent by that node, and *downstream* to refer to data received by that node. For instance, if a player sends and receives $N$ updates in every update period, it would have a bandwidth requirement of $NL_U/T_U$ in each of the upstream and downstream links. We note that the bandwidth requirement of the CS and PP architectures has also been studied in [4], under slightly different modeling assumptions, and experimentally measured for *Quake* [5].

### B. Inconsistency Resolution Latency

The *state* $S_i(t)$ of a game for player $i$ at a time instant $t$ is a collection of all attributes that describe completely the view of the game as player $i$ sees it at time $t$. Typically, the state of a game would include things such as position, direction, velocity and acceleration, carried objects, for each of the participating players.

Ideally, all players should have the same game state, i.e., $S_i(t) = S_j(t)$ for any pair of players $(i, j)$ and at any time instant $t$. Given that players communicate through a network, however, there is always some delay between players. Let $D_{i,j}$ be the network delay from player $i$ to player $j$. An action of player $i$ at time $t_0$ will become known to player $j$ at time $t_0 + D_{i,j}$. Thus, because of network delays, it is not possible to maintain state consistency among players at all times. Instead, we are interested in minimizing the time period during which two players have inconsistent states. We refer to this time period as *inconsistency resolution latency*. Consider, for instance, a CS architecture where the server detects and resolves all state inconsistencies among players. If player $i$ moves to position X at time $t_0$, while that position is already occupied by player $j$, a state inconsistency will result. The inconsistency will last until the server responds to player $i$ that its latest move is unacceptable. The inconsistency resolution latency in that case is related to the round-trip delay between the server and player $i$ (an exact formula is given in the next section).

The inconsistency resolution latency can vary for different players. So, we are mostly interested in the *Maximum Inconsistency Period (MIP)* $T_I$, which is the maximum inconsistency resolution latency among all players for a given game architecture. In deriving MIP, we assume that the network delays between players (or between a player and the server) have a known upper bound, and that update messages are never lost.

Previous work has proposed algorithms for maintaining state consistency in multiplayer games. A "bucket synchronization" mechanism was designed in [6] and used in the game MiMaze [7]. With bucket synchronization, updates sent by different players are processed at periodic time instants, referred to as "buckets", rather than immediately upon receipt. The latency between the receipt of an update and its processing intends to mask the variable delays between players. The bucket synchronization mechanism requires clock synchronization among players with an accuracy of a few tens of milliseconds. It is not clear whether the Internet clock

synchronization protocol NTP can provide such accuracy over wide-area networks.

[8] outlines several cheating techniques in multiplayer games, and proposes two mechanisms to avoid cheating. In the "lockstep protocol", players generate one-way hashes of their next move. Only after all players announce these next-move signatures, players will reveal their actual moves. An important issue about the lockstep protocol is that the game progresses at the pace of the slowest player. A less restrictive approach is the "asynchronous synchronization" protocol. Its difference with the lockstep protocol is a player only need to interact with players that reside in its "sphere of influence", similar to the filtering techniques described in [1], [9].

### C. Experimental Methodology

The experimental results reported in this paper were measured using *BZFlag*, a multiplayer open-source 3D tank battle game developed in 1992 by Riker [10]. The original implementation of *BZFlag* was based on the CS architecture. We have modified the code, however, to also create PP and PP-CA variations of the game.

The original BZFlag code did not provide state consistency. Player tanks could occupy the same space on the graphical terrain. We modified the game, so that the centralized server can resolve inconsistencies that are related to the position of player tanks. In particular, the server maintains the coordinates of all players. After player $i$ sends an update with its new coordinates, the server checks if the new position of player $i$ overlaps with a circle of a certain radius centered at each player's tank. If that is the case, the server denies player $i$'s move, returning to that player its previous coordinates.

Our measurements were performed on Pentium-III based PCs running Redhat Linux 7.0 connected via a Fast Ethernet Switch. Games were played between four (human) players for ten minutes, with the first 10,000 packets captured using "tcpdump" and used in our analysis. All players were located at the same lab, meaning that the network delays between the player machines were quite small (less than a millisecond). To facilitate the measurements, several modifications were made in the *BZFlag* source code, such as expanding application level headers to include timestamps on outbound packets, and adding additional server control messages to measure the inconsistency resolution latency.

The player update period and the server latency to respond to a player's update were measured through code profiling. Round-trip times were measured by bouncing a number of timestamped packets between the client and server. For the round-trip time measurements, the client and server operated in a tight-loop mode, returning packets immediately upon reception.

Inconsistency resolution latencies were measured as follows. The server checked received updates against the current coordinates of other players. If no conflict was found, the server permitted the move and forwarded the update to all players; otherwise, the server copied the timestamp from the received update to a corrected update, generated a position correction, and forwarded that revised update to all players. The player being corrected used the received timestamp to measure the inconsistency resolution latency.

## II. Client-Server Architecture

The most common architecture for multiplayer Internet games is the CS model. In this architecture, one host (sometimes a player itself) is designated as the server through which other players connect. The game simulation (i.e., state processing) takes place at the server, rather than at the clients. Each player sends its new actions to the server. The server uses the received information to advance the simulation, and it sends the resulting states back to the players who then display the updated game view to the users.

There are several reasons for the popularity of the CS architecture. In particular, with only one connection to worry about, it is a simpler architecture to implement. Additionally, it is simpler to maintain state consistency with a centralized server, and cheating becomes harder. Finally, game providers can create and control a subscription structure that they can potentially use for pricing and game monitoring.

There are some problems with the CS architecture however. First, the server represents a single point-of-failure [11], [12]. Second, while the bandwidth requirement at the players is minimal, the bandwidth requirement at the server can be significant [4], [12]. As a result, the server needs to operate behind a high capacity link. Additionally, relaying messages to players through a server adds communication latencies. These communication latencies can also increase the inconsistency resolution latency.

### A. Bandwidth Requirement

A client sends a player update to the server in every player update period, and so the client upstream bandwidth requirement is $L_U/T_U$. The downstream bandwidth requirement is derived from the updates sent from the server. Specifically, the server replies to each

player with a *global state* message, which represents the new state of each of the $N$ players. The size of the global state message is $L_G = NL_U$ bytes. So, the client downstream bandwidth requirement is $L_G/T_U$. The aggregate bandwidth requirement at a player in the CS architecture is

$$\frac{L_U + L_G}{T_U} = \frac{(N+1)L_U}{T_U} \quad (1)$$

which scales linearly with $N$.

The bandwidth requirement at the server can be derived similarly. The server receives player updates, advances the game state accordingly (for player updates that do not cause inconsistencies), and responds to the players with a global state message. The delay between receiving an update from a player and responding with a global state message to that player is referred to as *server latency* $T_S$. The upstream bandwidth requirement at the server is $NL_U/T_U$, because an update is received from each player in every update period. The downstream bandwidth requirement is $NL_G/T_U$, because a global state message is sent to each player in every update period. The aggregate bandwidth requirement at the server is

$$\frac{NL_U + NL_G}{T_U} = \frac{N(N+1)L_U}{T_U} \quad (2)$$

The key point here is that the bandwidth requirement at the server scales *quadratically* with the number of players $N$.

We note that, as it was shown in [9], the use of good *interest filtering techniques* can significantly reduce the network traffic at the server. Measurements taken at a *Quake* server showed that the bandwidth usage was less than predicted, even though it still scaled faster than linearly [4].

### B. Maximum Inconsistency Period

Recall that the MIP was defined as the maximum time period in which a player may have an inconsistent state, assuming no packet losses and bounded network delays. In the CS architecture, the MIP consists of three terms: the player update period $T_U$, the server latency $T_S$, and the round-trip time $R_{i,S}$ between player $i$ and the server $S$:

$$T_I = \max_i \{T_U + T_S + R_{i,S}\} \quad (3)$$

Note that we need to include the player update period, as it takes a full update period, in the worst case, from the time a message is received from the server to the time that that message is processed and displayed to the user.

### C. Experimental Results

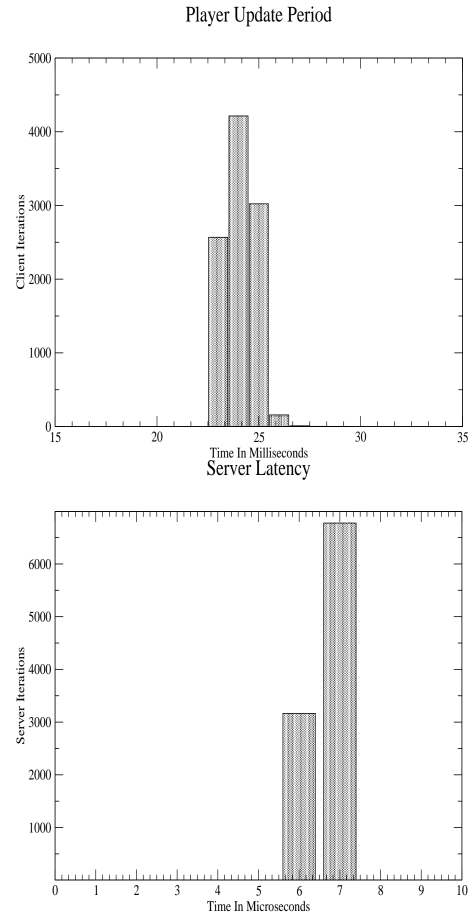


Fig. 1. CS game: player update period $T_U$ and server latency $T_S$.

Figures 1 and 2 show measurements of the three MIP components for a 4-player CS game. If we fill in the measurements of $T_U$, $T_S$, and $R_{i,S}$ in Equation 3, we find that the MIP for this game is about 27ms.

Figure 3 shows the inconsistency resolution latency measured for 5000 artificially caused inconsistencies in the previous 4-player CS game. The inconsistencies were introduced at the server, by manipulating the coordinates of the player tanks. We see that most inconsistencies are resolved within 25ms. The slight difference between the measurements of Figure 3 and the estimated MIP is that the latter is an upper bound of the inconsistency resolution latency, while the former are actual measurements of that latency.

## III. Peer-to-Peer Architecture

In the PP gaming model, each player (peer) is responsible for executing its own game simulation. Since there is no server to detect and resolve inconsistencies

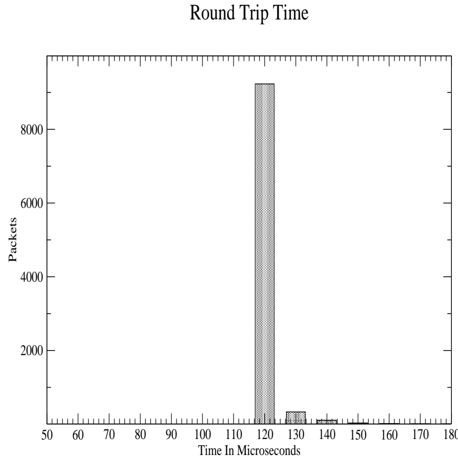


Fig. 2. CS game: round-trip time $R_{i,S}$ measurements

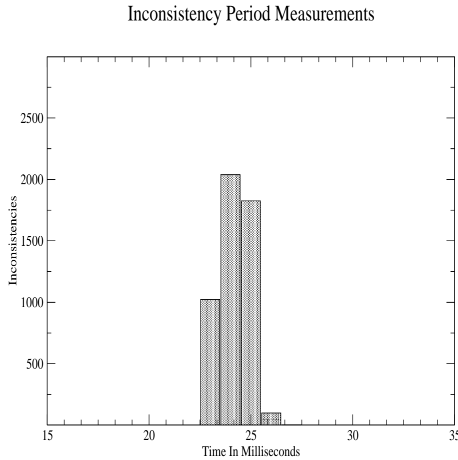


Fig. 3. CS game: inconsistency resolution latency measurements

among players, however, any inconsistencies have to be detected by the players using a distributed agreement protocol. For instance, the trailing state [13] or the bucket synchronization [7] techniques have been proposed to resolve inconsistencies in distributed games. An important issue with both these methods is that they rely on accurate clock synchronization. Even if player hosts run NTP (which is often not the case with home PC's), it is unclear whether NTP can synchronize clocks of machines connected through the commercial Internet with the required accuracy (typically, tens of milliseconds).

The lack of a centralized server brings several advantages to the PP architecture, such as reduced message latency between clients [11], [7], and elimination of a single point-of-failure. Perhaps most importantly, the PP architecture does not have a server bottleneck. While the PP architecture has a higher bandwidth requirement at the players than the CS architecture, that requirement increases linearly with the number of players.

The PP architecture is not nearly as popular, however, as the CS architecture in the gaming industry. An important reason is that the PP architecture is more difficult to implement and configure than a comparable CS architecture. Additionally, it is harder to maintain state consistency among players. Also, the PP architecture relinquishes too much control of the game to the users. The lack of a server means that there is no control over who is playing and for how long, and makes it harder for gaming companies to generate revenue via subscriptions.

### A. Bandwidth Requirement

In the PP architecture, each player sends and receives an update to and from each other players periodically. So, the *upstream* and *downstream* bandwidth requirements are the same, equal to $(N-1)L_U/T_U$. Thus, the aggregate bandwidth requirement at a player is

$$\frac{2(N-1)L_U}{T_U} \approx \frac{2NL_U}{T_U} \quad (4)$$

Note that this is roughly twice the bandwidth requirement of a player in the CS architecture. The bandwidth requirement, however, scales linearly with the number of players $N$, removing the server's bandwidth bottleneck. Interest filtering techniques such as those proposed in [9] can reduce the bandwidth requirement even further.

### B. Maximum Inconsistency Period

In a PP architecture, an inconsistency can be detected when a player $j$ receives an update from player $i$ which disagrees with player $j$'s state. In that case, player $j$ responds to all players, including player $i$, with a "update-rejection" message. The purpose of that message is to invalidate $i$'s latest update. Upon the receipt of that message, player $i$ needs to rollback to its previous (accepted) state. The inconsistency resolution latency in this case would consist of the round-trip delay between players $i$ and $j$ $R_{i,j}$, and of twice the update period (one $T_U$ spent in the worst-case at player $j$ and another $T_U$ spent at player $i$). Thus, the maximum inconsistency resolution period for any two players would be:

$$T_I = \max_{i,j}\{2T_U + R_{i,j}\} \quad (5)$$

assuming bounded network delays and no packet losses.

Whether the inconsistency resolution latency of the PP architecture is less than that of the CS architecture depends on the relation between the round-trip times $R_{i,j}$ and $R_{i,S}$.

## IV. Peer-to-Peer with Central Arbiter Architecture

A game architecture with the scalability properties of the PP model, but also with the simple consistency resolution mechanism of the CS model, would be desirable. To that end, we propose a modifi ed version of the PP model which we call *Peer-to-Peer with Central Arbiter* model (PP-CA).

In PP-CA, players exchange updates communicating directly with each other, just as in the PP model. This minimizes the communication delays between players. Each player sends its updates not only to all other players, but also to the central arbiter. The role of the central arbiter is to listen to all player updates, simulate the global state of the game, and detect inconsistencies. In the absence of inconsistencies, the central arbiter remains silent, without sending any messages to the players. When an inconsistency is detected however, the central arbiter will resolve it, create a corrected update, and transmit that update to all players. The corrected players should then rollback to the previous accepted state.

The consistency resolution protocol in PP-CA is basically the same with that in the CS architecture. The key difference between the CS server and the PP-CA arbiter is that the former sends a global state update (of size $NL_U$) to each of the $N$ players in every player update period, while the latter sends a corrected update (of size $L_U$) to each of the $N$ players only when an inconsistency occurs. If inconsistencies are rare events, the bandwidth requirement at the PP-CA arbiter will be signifi cantly lower than the bandwidth requirement at the CS server.

### A. Bandwidth Requirement

In the downstream direction, the bandwidth requirement at a PP-CA player is $(N-1)L_U/T_U$, because a player receives updates from (N-1) other players in each player update period. The bandwidth requirement can be higher by $L_U/T_U$, however, when the central arbiter detects a state inconsistency. In the upstream direction, we need to include the central arbiter as an additional peer, and so the bandwidth requirement is $NL_U/T_U$. The aggregate bandwidth requirement at a PP-CA player, in Equation 6, shows a slight increase compared to the PP model, but the relation with the number of players $N$ is still linear,

$$\frac{NL_U + (N-1)L_U}{T_U} \approx \frac{2NL_U}{T_U} \quad (6)$$

The bandwidth requirement at the central arbiter depends on the frequency of state inconsistency events. If inconsistencies never occur, the central arbiter just receives $N$ updates in each update period, without sending any messages, and its aggregate bandwidth requirement becomes $NL_U/T_U$. If a single state inconsistency occurs in each update period, the central arbiter would have to send $N$ corrected updates in each update period. In that case, the aggregate bandwidth requirement at the central arbiter would be $\frac{2NL_U}{T_U}$, which is equal to the bandwidth requirement at a player, and scales linearly with $N$. The bandwidth requirement would be higher if multiple inconsistency events occur in the same update period. The worst case scenario is that each player reports an inconsistency in every update period; in that cawe the bandwidth requirement at the central arbiter would increase quadratically with $N$, as in the server of the CS architecture.

### B. Maximum Inconsistency Period

In the PP-CA model, inconsistencies are detected and resolved as in the CS model. Thus, the MIP is

$$T_I = \max_i \{T_U + T_S + R_{i,S}\} \quad (7)$$

### C. Experimental Results

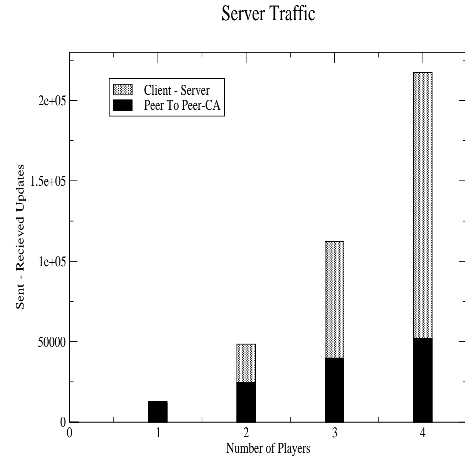


Fig. 4. CS vs PP-CA server bandwidth requirement.

Measurements of an active BZflag game, shown in Figure 4, compare the traffi c load at a CS server and at a PP-CA central arbiter. In this experiment, we have artifi cially created a 50% inconsistency rate at the exchanged updates (i.e., 50% of the player updates received at the central arbiter are detected as inconsistent). The traffi c load at the central arbiter increases almost linearly with the number of players, and it is signifi cantly lower than the traffi c load at the CS server which increases quadratically.

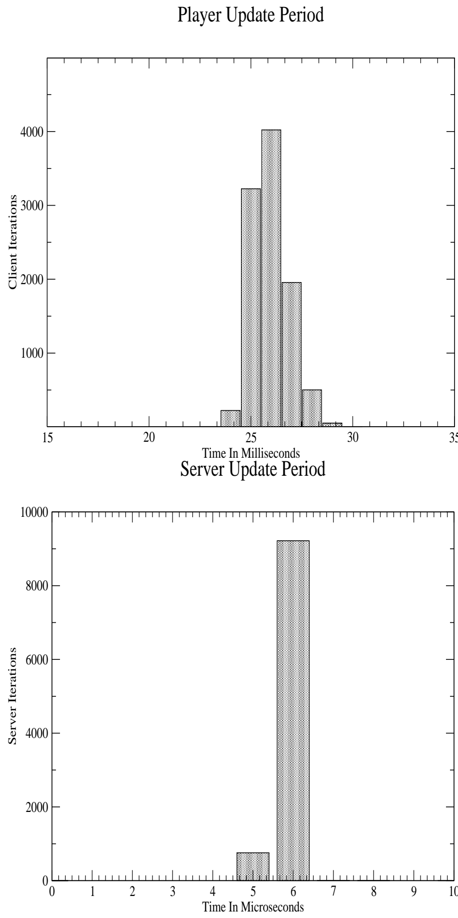


Fig. 5. PP-CA game: player update period $T_U$ and server latency $T_S$.

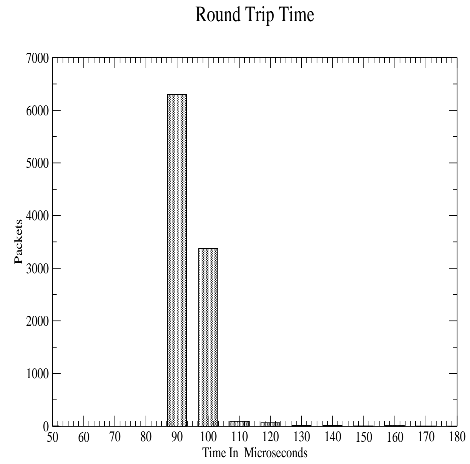


Fig. 6. PP-CA game: round-trip time $R_{i,S}$ measurements

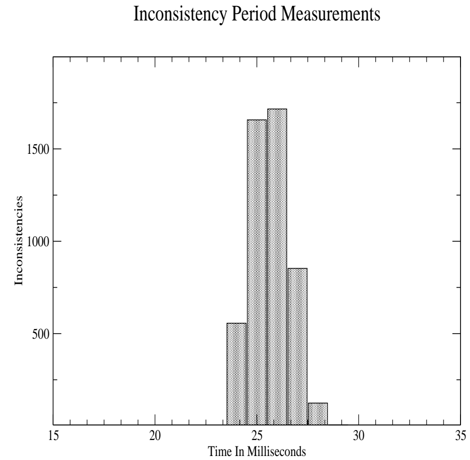


Fig. 7. PP-CA game: inconsistency resolution latency measurements

Figures 5 and 6 show measurements of the three MIP components for a 2-player CS game. If we fill in the measurements of $T_U$, $T_S$, and $R_{i,S}$ in Equation 7, we find that the MIP for this game is roughly 29ms.

Figure 7 shows the inconsistency resolution latency measured for 5000 artificially caused inconsistencies in the previous 2-player CS game. We see that most inconsistencies are resolved within 28ms. The slight difference between the measurements of Figure 7 and the estimated MIP is that the latter is an upper bound of the inconsistency resolution latency, while the former are actual measurements of that latency.

## V. Conclusions

The PP-CA model attempts to combine the best features of the CS and PP architectures. The player-to-player communication latency is lower in the PP architecture, and this feature is inherited to the PP-CA architecture. Inconsistencies in PP-CA are resolved by a central entity (arbiter), and so a complex distributed agreement protocol is not required. The central arbiter receives updates from all players, but it only sends corrected updates when an inconsistency is detected. The bandwidth requirement at the central arbiter increases linearly with the number of players, and it is equal to the player bandwidth requirement, in the case of a single inconsistency per update period. Additionally, the presence of a centralized arbiter allows game providers to still monitor the game for accounting or pricing reasons.

Finally, we note that even though the PP-CA helps to reduce the bandwidth requirement of the central arbiter, it does not reduce its processing load. The central arbiter needs to simulate the state of the entire game, just like a CS architecture server. This means that if the bottleneck of a game is the CPU power of the server, rather than its network bandwidth, the PP-CA architecture will not be scalable either.

## References

[1] J. Smed and T. Kaukoranta and H. Hakonen "Aspects of Networking in Multiplayer Computer Games". In Proceedings of International Conference on Applications and Development of Computer Games in the 21st Century, November 2001.

[2] L. Pantel and L. C. Wolf. "On the Impact of Delay on Real-Time MultiPlayer Games". In Proceedings of ACM NOSSDAV, May 2002.

[3] G. Coulouris, and J. Dolimore, and T. Kindberg. "Distributed Systems Concepts and Design". Addison-Wesley, 2001.

[4] E. Cronin, B. Filstrup, and A. Kurc. "A Distributed Multi-Player Game Server System". EECS589, Course Project Report, University of Michigan, May 2001.

[5] ID Software. "Quake". Available at http://www.quake.com, 2002.

[6] L. Gautier and C. Diot and J. Kurose. "End-to-End Transmission Control Mechanisms for Multiparty Interactive Applications on the Internet". In Proceedings of IEEE INFOCOM, April 1999.

[7] C. Diot and L. Gautier. "A Distributed Architecture for MultiPlayer Interactive Applications on the Internet". In IEEE Network magazine, 13(4), August 1999.

[8] N. Baughman and B. Levine. "Cheat-Proof Playout for Centralized and Distributed Online Games". In Proceedings of IEEE INFOCOM, April 2001.

[9] L. Zou, M. H. Ammar, and C. Diot. "An Evaluation of Grouping Techniques for State Dissemination in Networked Multi-User Games". In Proceedings of MASCOTS, August 2001.

[10] T. Riker. "BZFlag". Available at http://www.bzflag.org, 2002.

[11] P. Bettner and M. Terrano. "1500 Archers on a 28.8 Programming in Ages of Empires and Beyond". Technical report, Ensemble Studios, 2001.

[12] M. Mauve. "How to Keep a Dead Man from Shooting". In Proceedings of 7th International Workshop on Interactive Distributed Multimedia Systems, October 2000.

[13] E. Cronin, B. Filstrup, and A. Kurc and S. Jamin. "An Efficient Synchronization Mechanism for Mirrored Game Architectures". In Proceedings of ACM NETGAMES, April 2002.