An Efficient Synchronization Mechanism for Mirrored Game Architectures
(Extended Version)

Eric Cronin, Anthony R. Kurc, Burton Filstrup and Sugih Jamin*
({ecronin,tkurc,bfilstru,jamin}@eecs.umich.edu)
Electrical Engineering and Computer Science Department
University of Michigan

Abstract. Existing online multiplayer games typically use a client-server model, which introduces added latency as well as a single bottleneck and single point of failure to the game. Distributed multiplayer games minimize latency and remove the bottleneck, but require special synchronization mechanisms to provide a consistent game for all players. Current synchronization methods have been borrowed from distributed military simulations and are not optimized for the requirements of fast-paced multiplayer games. In this paper we present a new synchronization mechanism, *trailing state synchronization* (TSS), which is designed around the requirements of distributed first-person shooter games.

We look at TSS in the environment of a *mirrored game architecture*, which is a hybrid between traditional centralized architectures and the more scalable peer-to-peer architectures. Mirrored architectures allow for improved performance compared to client-server architectures while at the same time allowing for a greater degree of centralized administration than peer-to-peer architectures.

We evaluate the performance of TSS and other synchronization methods through simulation and examine heuristics for selecting the synchronization delays needed for TSS.

Keywords: Consistency, Game Platforms, System Architectures

1. Introduction

Online multiplayer games typically take one of two basic forms: centralized client-server (all commands go through a single server), shown in Fig. 1.a, or distributed peer-to-peer (commands go directly to other players), shown in Fig. 1.b. Client-server architectures are usually simpler, but are also less efficient and scalable. Every command must go from the client to the server and then be re-sent by the server to other

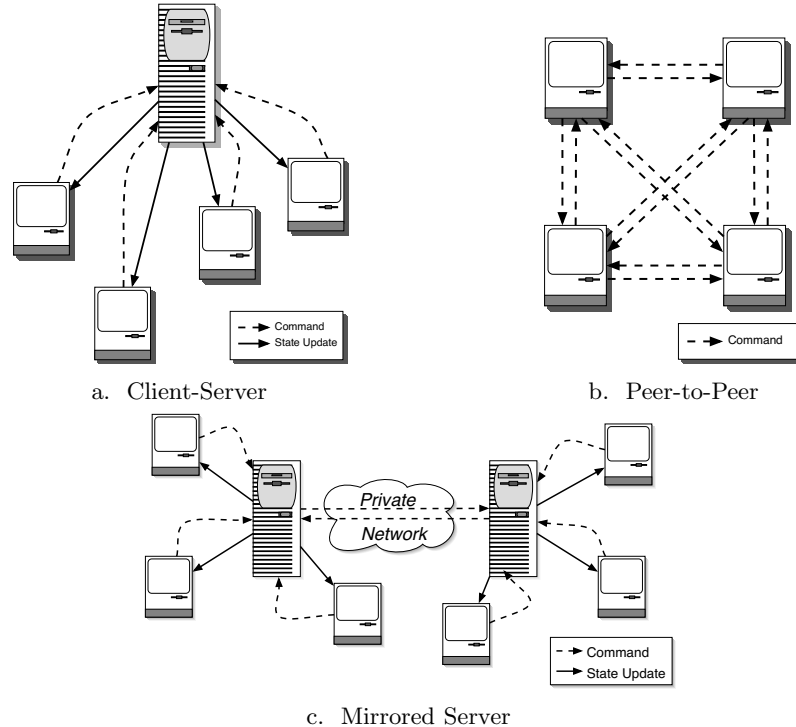a. Client-Server   b. Peer-to-Peer

c. Mirrored Server

Figure 1. Multiplayer Game Architectures.

clients in the form of update messages. This adds additional latency over the minimum cost of sending commands directly to other clients. In addition, the server becomes a single point of failure in the game. Unlike centralized games, where there is a single authoritative copy of the game state kept at the server, distributed game architectures require a copy of the entire game state to be kept at each client. As a result, these architectures require some form of synchronization between clients to ensure that each copy of the game state is the same. Without synchronization, clients' game states would diverge over time due to network delays and other factors.

Common synchronization techniques used in existing distributed simulation environments include bucket synchronization, breathing bucket synchronization, and TimeWarp synchronization [8, 19]. Many of these synchronization mechanisms were initially designed for use in large-scale military simulations, and were only later adapted for use in multiplayer games as games gained popularity. In this paper we present a novel new synchronization mechanism, *trailing state synchronization* (TSS), which is designed specifically with distributed first-person shooter games such as Quake [10] in mind. First-person shooters are the

most latency-sensitive class of multiplayer games, and have a different set of optimization parameters than large military simulations. Existing synchronization techniques suffer from problems such as introducing too much additional latency, providing only loosely consistent synchronization, or requiring high memory and computation costs. TSS is designed to execute commands quickly while at the same time maintaining a consistent copy of the game state at all players, and doing this in a computationally and memory-wise efficient manner.

The remainder of the paper is organized as follows: in Section 2 we provide background on multiplayer game architectures, including the mirrored game architecture, and previously proposed synchronization methods. In Section 3 we introduce trailing state synchronization. Section 4 examines the performance of TSS and compares it with other synchronization algorithms. Finally, Section 5 concludes.

2. Background

2.1. Mirrored Game Server Architecture

Despite the added latency, the single point of failure, and the scalability problems, the client-server architecture is by far the most commonly used architecture in current multiplayer games. There are a number of reasons behind the popularity of the client-server architecture. First, the networking code is simpler to write since complicated synchronization processes can be avoided. Often a single player version of a game can be quickly adapted for client-server play with only minor changes. Second, and usually more importantly, controlling the server gives the game publisher more administrative control. Having control over game servers lets publishers perform authentication, copy protection, accounting and billing, and easy update of client code. Third, to reap all the benefits of a peer-to-peer architecture, a multicast connection between clients is needed to reduce the bandwidth requirements. Unfortunately, IP multicast is not yet widely available, and most existing peer-to-peer games resort to sending a separate copy of each message to every player, greatly increasing the bandwidth requirements [13]. End-host multicast [9] can be used to reduce this problem, but to our knowledge has not been used in multiplayer games.

The *mirrored server architecture* (Fig. 1.c) that we first proposed in [4] is a hybrid architecture designed to address the problems with client-server and peer-to-peer architectures, similar to the generic proxy architecture proposed in [15]. Instead of a single central server, there are multiple distributed servers for each game. Clients connect to the mirror

closest to themselves in a traditional client-server fashion. Players can either pick a mirror manually, or the game client could make use of an Internet distance service [7] to automatically select a mirror close to the client. If the mirrors are well placed, the additional latency overhead of the client-server architecture is greatly reduced. The mirrors themselves are then connected to each other over a private, low-latency multicast network used only for game traffic. The mirrors exchange commands using a peer-to-peer architecture, with each mirror maintaining its own copy of the game state. The use of a private network allows IP multicast to be realistically used.

Since there are now multiple servers for the same instance of a game, the single point of failure in traditional client-server architectures is eliminated. If any one of the servers crashes, the clients connected to it will be disconnected, but the other servers and clients can continue with the game. Unlike peer-to-peer games, the networking complexity in the mirrored server architecture lies in the servers not in the client. In fact, in our prototype mirrored server version of Quake described in Section 4.1, the Quake client itself is not changed at all. Unlike a fully distributed game architecture, the mirrored servers are still under the game publisher's control. This allows for authentication, copy protection, as well as the ability to trust mirrors. The ability to trust the mirrors is important because in peer-to-peer architectures (which the mirror architecture is) where the clients are untrusted, it becomes easier to cheat, and it becomes more difficult for other participants to detect when someone is cheating [2].

The use of mirrored servers does place some restrictions on what the synchronization algorithm can do. Each mirror must be able to handle multiple clients that could be located in any part of the game's world at any given time. As a consequence, it is not straightforward to perform *interest management* between mirrors. In a purely peer-to-peer architecture, two clients who are not interacting with each other need not be tightly synchronized, reducing the bandwidth requirements, and preventing clients from knowing information they do not absolutely need (thus reducing the opportunities for cheating). A limited degree of interest management between individual clients and their mirror is still possible, e.g., the mirror can send clients only the information necessary to render their current locations. We plan to study this issue in greater detail as part of our future work.

2.2. Synchronization Techniques

In peer-to-peer games, instead of sending commands to a central server that computes the game state and issues updates, clients send messages

directly to each other. In order for each client to have a consistent view of the game state, there needs to be some mechanism to guarantee a global ordering of events[1] [12]. This can either be done by preventing misordering outright (by waiting for all possible commands to arrive), or by having mechanisms in place to detect and correct misorderings. An additional complication to synchronization of multiplayer arises from their continuous nature. Even if there are no direct misorderings of commands, the time at which they are executed can create additional ordering constraints. In these simulations, both the order and the simulation time of execution are important in maintaining consistency. How this synchronization is performed is very important to the success of the game; if it is not possible to maintain ordering within a reasonable time, no one will be willing to play the game.

2.2.1. *Conservative Algorithms*

Lockstep synchronization [17], used in military simulations, is by far the simplest technique available to ensure consistency. No member is allowed to advance its simulation clock until all other members have acknowledged that they are done with computation for the current time period. This takes the first approach to providing a global ordering of events: preventing out of order events from even being generated. In this system, it is impossible for inconsistencies to occur since no member performs calculations until it is sure it has the exact same information as everyone else. Unfortunately, this scheme also means that it is impossible to guarantee any relationship between simulation time (also sometimes referred to as *global virtual time (GVT)* or *game time*) and wall-clock time. There is no way to guarantee that the game will advance at a regular rate, much less at a rate fast enough for interactive gameplay. In a multiplayer game, this is not an acceptable tradeoff. There are a number of similar conservative algorithms that perform better than lockstep but are still unable to maintain a constant rate in all situations. One of these is fixed time-bucket synchronization [17], where a synchronization delay is used to reduce the dependency on latency between members; an optimistic version of this algorithm that is better suited to multiplayer games is discussed below. While these "conservative" algorithms perform poorly in fast-paced games where a constant rate of simulation is important, they may still be suitable for slower turn-based games.

---

[1] When discussing TSS we often use *events* and *commands* interchangeably, as the events being synchronized in Quake are player commands.
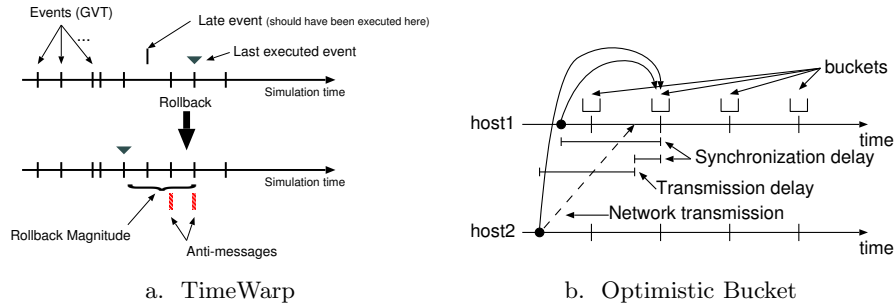
Figure 2. Optimistic Synchronization Algorithms.

2.2.2. *Optimistic Algorithms*

The second approach to ensuring consistency is to detect and correct any differences in states. Optimistic algorithms execute events before they know for sure that no earlier events could arrive, and then repair inconsistencies when they are wrong. This class of algorithms is far better suited for interactive situations. It is worth looking at several examples of existing optimistic algorithms and their shortcomings when used with games like Quake before describing TSS.

TimeWarp synchronization [19] (Fig. 2.a) works by taking a snapshot of the state before each execution (we call this snapshot the *execution context*), and issuing a *rollback* to an earlier state if an event earlier than the last executed event is received. On a rollback, the state is first restored to that of the snapshot, and then all events between the snapshot time and the current execution time are re-executed, including the late command in its correct place. Periodically, all members reset the oldest time at which an event can be outstanding, thereby limiting the number of snapshots needed (and also limiting the maximum lateness of a command that can be recovered from).

A main limiting feature of TimeWarp synchronization in a game such as Quake is the requirement to checkpoint the execution context at every message. A Quake execution context consumes about one megabyte of memory, and new messages arrive at a rate of one every thirty milliseconds from each client (more recent games have even higher frame rates and larger execution contexts). Additionally, copying an execution context involves not just the memory copy but also repairing linked lists and other dynamic structures. Copying state on every command requires both a fast machine and large amounts of memory.

One optimization to remove this limitation is to only take snapshots of the state periodically [14], which reduces the memory and copy overheads, but makes rollbacks potentially more costly since there

is no longer always a snapshot from exactly before the inconsistency occurred. Additionally, to reduce memory usage, the snapshots can contain only what has changed in the state instead of the entire state. This enhancement however complicates both command execution (requiring all changes to state to be explicitly logged) and rollbacks (requiring the game to support an "undo" mode of operation), both of which require tight integration into the design of the game engine.

Another complication with rollbacks is that TimeWarp assumes that events directly generate new events. As part of the rollback, special *anti-messages* are sent out to cancel previously generated events that have become invalid (that in turn can trigger other rollbacks if these messages have already been executed, which in turn trigger more anti-messages and so on). This explosion of anti-messages can bog down the network and tie up servers with anti-message processing instead of executing the game. In addition, in interactive games the player sits between incoming events and newly generated ones, removing this direct causal relationship. Anti-messages do not fit in well with this architecture, since the player is outside the control of the synchronization mechanism.

Finally, TimeWarp issues a rollback immediately upon detecting a late command. While this has the benefit of making rollbacks as small as possible, if there are several delayed commands in rapid succession, there will be several of these rollbacks in a row. The combined effect of these multiple rollbacks is similar to that of a single larger rollback to the player. If the additional delay is small, delaying these rollbacks and aggregating them into one will not be noticed by the player.

"Breathing" algorithms [18] attempt to solve the problem of excessive rollbacks seen in TimeWarp by restricting the number of commands that can be executed optimistically. Instead of fully optimistic execution, breathing algorithms limit their optimism to events within an *event horizon.* Events beyond the horizon can not be guaranteed to be consistent, and are therefore not executed. As with anti-messages in TimeWarp, a problem with applying this to Quake is that nearly all events in Quake are not directly generated by other events. Instead, they are generated by user actions (that may be influenced by earlier events), and it is therefore not clear how to accurately define an event horizon.

The algorithm implemented in MiMaze [8] is an optimistic version of the conservative bucket synchronization algorithm (Fig. 2.b). Events are delayed for a time that should be long enough to prevent misorderings before being executed. If events are lost or arrive later than expected, however, MiMaze does not attempt to detect inconsistencies or recover in any way. If no events from a member are available in

a particular bucket, the previous bucket's event is *dead reckoned*, for example by simply replaying the last event again or extrapolating a new event based on the last few events; if multiple events are available in a particular bucket, only the most recent one is used. Late events are scheduled for the next available bucket; however, late events are not likely to be used because only one event for each member is executed per bucket. For a simple game such as MiMaze, in which movement is limited to a confined maze where positional errors are minimal, and interactions between players are limited enough that any inconsistencies do not dramatically impact gameplay, these optimizations at the cost of consistency may be acceptable. For a game like Quake, where interactions between players are much more frequent, small inconsistencies are likely to combine to lead to larger divergences between different states. It is possible to decrease the number of inconsistencies in bucket synchronization, but only by increasing the synchronization delay, which decreases responsiveness.

3. Trailing State Synchronization

None of the existing distributed game or military simulation synchronization algorithms introduced above are entirely suited to a game such as Quake that has frequent updates, has a need for strong consistency, and is very latency sensitive. Our solution to this problem is a new algorithm called trailing state synchronization (TSS). Similar to TimeWarp, TSS is an optimistic algorithm, and must execute rollbacks when inconsistencies are detected. However, it does not suffer from the high memory or processor overheads of TimeWarp, is able to aggregate multiple rollbacks into a single one, is able to provide an arbitrary maximum lateness that can be recovered from, and has the potential to recover from inconsistencies without full rollbacks in some situations.

When rollbacks are required, instead of copying the state from a snapshot taken just prior to the offending command as TimeWarp does, TSS copies the state from a second copy of the same game that is running at a delay relative to the inconsistent state. This second copy of the game state, since it is *trailing* the first in execution, has had more time to reorder commands and does not have the inconsistency that must be repaired (in fact, as we will see later, it is this second state that actually detects that an inconsistency has occurred). This use of dynamically changing states as the source of rollbacks as opposed to static (including periodic [14]) snapshots is a fundamental difference between TSS and TimeWarp.
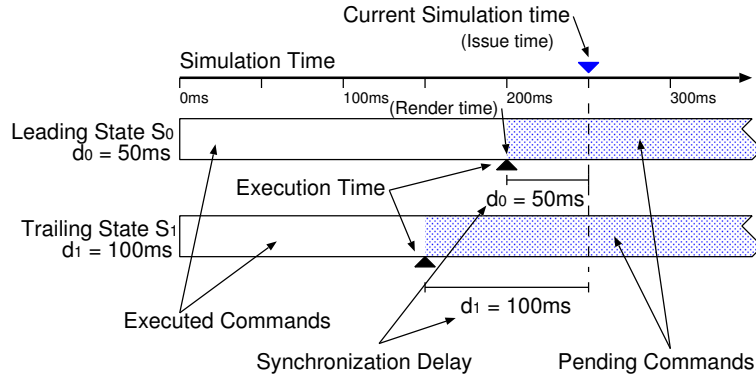
Figure 3. TSS Terminology.

Instead of keeping snapshots at every command (or every few commands), TSS keeps and updates a fixed number of copies of the game state, each of which is at a different simulation time. These copies, referred to as *states*, each execute every command, but after differing synchronization delays. Only the *leading state*, which has the shortest synchronization delay, is rendered to the clients' screens, while the other *trailing states* are used to detect and correct inconsistencies. At simulation time t, if the leading state S_0 is executing a command from simulation time $t - d_0$, then the first trailing state S_1 will be executing commands up to simulation time $t - d_1$, the second trailing state S_2 commands up to $t - d_2$ and so on, where $d_0 < d_1 < d_2 < \ldots$. In this manner, only one snapshot's worth of memory is required for each trailing state, reducing and bounding the memory requirements. Note that the synchronization delays do not have to be linearly spaced. From this description, it is clear that a main assumption with TSS is that the cost of executing commands multiple times in each of the states is less expensive than just taking a snapshot. In Section 4 we will revisit this assumption, and see that for first person shooters it appears to hold true.

Fig. 3 graphically depicts the components of TSS. Time increases to the right. The 0 point on each state is in reference to execution time, not simulation time. TSS is able to provide consistency because each trailing state sees fewer misordered commands than the state preceding it by waiting longer for delayed commands to arrive before executing. The leading state executes with a small synchronization delay (d_0). The synchronization delay for a state is defined as the difference between the current *simulation time* (a global value at each client with which new commands are timestamped before being sent) and the *execution time* of the state. The simulation time is used to allow commands to

be reordered properly before execution. If a command is stamped with simulation time t at the generating client, then it cannot be displayed until simulation time $t + d_0$, even at that same client. In other words, the simulation time is used internally to timestamp commands when issued, while the execution time of the leading state is what is actually rendered to clients.

3.1. TSS Operation

As commands arrive (or are generated) at a client, they are placed on a pending list for each trailing state in timestamp order. Late moves whose timestamps are earlier than the current execution time for a state are placed at the head of the pending list for the state and are executed immediately. Different copies of a particular command in each trailing state are linked together, so that the command from state S_i can find the copy of the same command that was executed in the preceding state S_{i-1}. To detect inconsistencies, each trailing state looks at the changes in game state that the execution of a command produced, and compares them with the changes recorded in the directly preceding state. While this execution information kept is similar to that kept in the second TimeWarp optimization discussed in Section 2.2.2, the record in TSS need not contain everything that would be required to repair the changes, but only enough data to determine that the two executions were not identical. This has the potential to make TSS simpler to integrate into an existing game engine.

When a command is executed in the last of the trailing states, the command is deleted from all states since it will never be needed again. The last state has no trailing state to synchronize it, and therefore any inconsistencies in it will go undetected. However, if it is assumed (as is done in the other bounded optimistic algorithms) that the longest synchronization delay is large compared to expected command transit delays, this is unlikely to pose a problem.

If an inconsistency is discovered, a rollback from the incorrect leading state to the correct trailing state is performed in the leading state. This consists of copying the game state from the trailing state to the leading state, as well as adding back to the leading state's pending list any commands that were executed in the incorrect state after the rollback time. The next time the leading state executes, it will re-execute these commands and return to the proper execution time for its synchronization delay. If a rollback occurs between states S_i and S_{i-1}, it is possible that once S_{i-1} re-executes to correct the inconsistency, new inconsistencies between S_{i-1} and S_{i-2} will be found, generating a

rollback between S_{i-1} and S_{i-2}. In this fashion, any inconsistencies in a trailing state that the leading state also shares will be corrected.

When an inconsistency is detected between states S_i and S_{i-1}, where $i > 1$, the rollback "cascades" until it reaches S_0. The first rollback will be between S_i and S_{i-1}. Then, once S_{i-1} re-executes and detects the inconsistency with *its* leading state S_{i-2}, a second rollback between S_{i-1} and S_{i-2} will occur. This repeats until finally S_0 is corrected. This mechanism is used as opposed to just executing several rollbacks immediately between S_i and S_{i-1}, S_i and S_{i-2}, ... , S_i and S_0 to prevent unnecessary re-execution. In order for TSS to continue to function correctly, all the states between S_i and S_0 must eventually re-execute commands between S_i's execution time and the execution time of the intermediate state to maintain consistency. If the rollbacks all occur at once, many of these re-executions will be duplicated in all the states, whereas with cascading rollbacks each command is only re-executed once and the updated (consistent) state copied to the preceding state where further re-executions will be done. There is no additional cost to using cascading rollbacks other than having to do the comparison to detect that an inconsistency has occurred in each state.

One of the not immediately obvious features of the way TSS detects inconsistencies is that a rollback in TSS may actually repair more than one inconsistency, eliminating the need for future rollbacks. In Time-Warp, as soon as an out of order command is received, a rollback occurs. In TSS however, rollbacks are delayed until the trailing state reaches the point where the inconsistency occurred in the leading state. In this extra time, other commands may have arrived which were also late for the leading state, creating more inconsistencies. When the rollback occurs however, these commands will be moved back to the pending list to be re-executed, this time in the correct order. This feature can be seen as delaying the rollback in order to amortize its cost over several inconsistencies. This amortization of rollbacks is unique to TSS among the optimistic synchronization algorithms studied.

When a rollback occurs, the player's position will jump from the incorrect position to the correct position and gameplay continues. Occasionally, rollbacks will cause more drastic changes, such as the player coming back to life when they thought they had been killed. The impact of these events can be lessened by delaying slightly the notification that the player has been killed, in case there is a rollback. Instead of issuing anti-messages on rollbacks as in TimeWarp, we take the same approach often used in dead reckoned games, applying the new commands to the corrected state. These problems are no worse in TSS than in any other dead reckoned game that corrects for inconsistencies [3].

3.1.1. *Weakly and Strictly Consistent Events*

The synchronization techniques studied in Section 2.2 treat all commands in the same way, i.e., they do not distinguish between different types of commands when determining how to handle late commands. By differentiating between different types of commands, TSS has the potential to avoid rollbacks in some situations where TimeWarp (or other synchronization algorithms that do not employ multiple execution states) would not be able to. By waiting until the results of the on-time execution of a command are known before triggering a rollback, TSS has more information available when determining if the rollback is necessary. If either the correct or incorrect execution result is unavailable when the rollback decision must be made, as is the case with TimeWarp, the only choice is to be conservative and trigger a rollback.

In Quake, there are two basic classes of commands. The first type we refer to as *weakly consistent*, which consists of move commands. With these commands, it is not essential that the same move happened at the same time, as much as that the position of the player in question is within some small margin of error in both states. The other class of commands we refer to as *strictly consistent*. For these commands, such as weapons being fired (particularly projectiles), it is important that both states agree on exactly when and where the event occurred. Strictly consistent commands also dominate any weakly consistent commands near them in time. For example, when a weapon is fired, having the exact correct position for players within the weapon's range becomes important. For other uses of TSS, there may be similar classifications where comparing the results of a command, as opposed to just the execution time of the command, gives a better indication of whether an inconsistency occurred.

The advantage of separating commands into classes in this manner is that when comparing the results of executing a command in two states, weakly consistent commands can be allowed a margin of error, and less invasive means of repairing inconsistencies can be employed (for example, by adjusting the player's *trajectory* to correct for positional errors). If these less invasive corrections are ineffective, the error will become large enough to trigger a rollback, to maintain complete consistency. The disadvantage is that the game must now provide functions to classify commands, compare two executions of commands, and perform corrections without rollbacks.

If commands are not separated into classes, the information needed to determine if two executions of a command are consistent is reduced to determining if the command was executed on time or not. This has the advantage that if a move arrives late, there is no need to place it
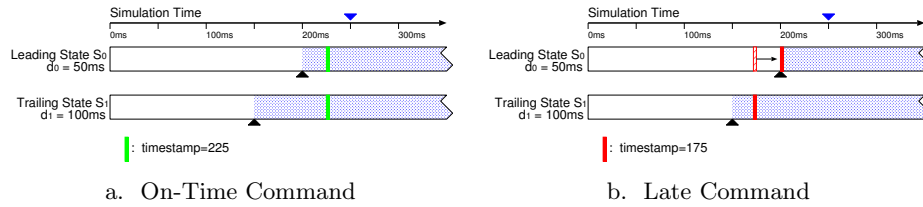
a. On-Time Command   b. Late Command

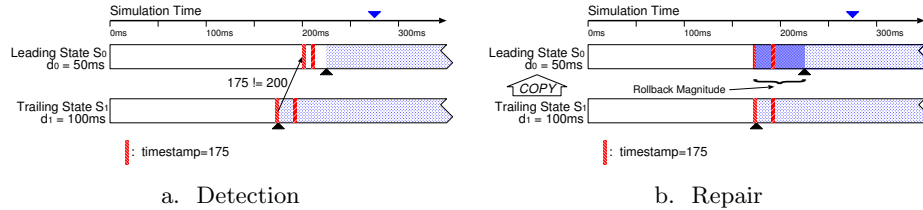*Figure 4.* TSS Execution.

a. Detection   b. Repair

Figure 5. TSS Rollback.

at the head of the pending list to be executed immediately, since we already know that this execution will be incorrect and it will need to be re-executed once a rollback occurs. Depending on the game however, it may still be advantageous to execute the command immediately, even though we know it will be re-executed later, in order to lessen the impact of the rollback on the client.

3.2. AN EXAMPLE OF TSS

We provide a graphical example of TSS's operation. Fig. 4.a shows the normal on-time operation of TSS. A command with timestamp 225ms arrives at the synchronizer when it is at simulation time 250ms. Because the synchronization delays in use are 50ms and 100ms, the command is placed in the pending list for both states, to be executed in the future. It will first be executed at simulation time 275ms in state S_0. At time 325ms, state S_1 will execute it, and compare its results with that of S_0. Since they are identical, no inconsistency is found and the command is discarded from the synchronizer.

Fig. 4.b depicts what happens when a command arrives late. The synchronizer is still at simulation time 250ms, but now the arriving move has a timestamp of 175ms. The state S_0, which has a synchronization delay of 50ms is already at execution time 200ms, so the command is executed immediately there, but still 25ms later than it should have been (we assume for illustrative purposes that late commands are always executed). State S_1 has a large enough synchronization delay and the move can still be executed on time.

When S_1 executes this command and compares its result with that of S_0, they do not match (Fig. 5.a), which triggers a rollback (Fig. 5.b). The entire state of S_1 is first copied over S_0's incorrect state. Any commands in the 50ms gap between where S_1 has executed up to and where S_0 was executing before the rollback (dark shaded region in the figure) are moved back to the pending list in S_0 to be re-executed. If any of the other commands in the dark shaded region had originally arrived late at S_0 (e.g. the second command shown, with timestamp 290), they will no longer generate a rollback in the future since they will be re-executed on-time as part of the 175ms command's rollback.

3.3. Analysis of TSS

TSS is an optimistic algorithm, executing commands before it is certain that no earlier commands may still be in transit. This distinguishes it from the entire family of conservative protocols, and makes it suitable for use in first-person shooters where the game must be able to keep up with wall-clock time. Although it uses a synchronization delay as in MiMaze, TSS is able to provide both a quick initial delay and a long window for which all inconsistencies will be detected through the use of multiple states. In MiMaze, these two variables are controlled through the single synchronization delay parameter, so the system can either be responsive or consistent but not both.

TSS is most similar to TimeWarp, since both are fully optimistic and employ rollbacks to recover from any inconsistencies. With the addition of optimizations such as less frequent snapshots, to contain the memory usage, and a synchronization delay, to prevent unnecessary rollbacks in a game environment, TimeWarp can be made to perform similarly to TSS in the respective aspects. TimeWarp is not, however, able to handle bursts of late events in a single rollback, nor is it able to control how much re-execution is needed when fewer snapshots are used (sometimes a rollback will occur a few events after a snapshot, sometimes a few events before the next snapshot). In addition, TimeWarp is unable to take advantage of event classifications to reduce the number of rollbacks needed.

TSS performs best when the cost of executing a command (multiple times) is less than the cost of making a snapshot because it must run multiple parallel versions of the same game. Experiments in Section 4 show that in the case of Quake, the cost of executing a command is an order of magnitude less than the cost of making a snapshot. Although it must execute each command multiple times (once for each state being maintained), since each state is independent, TSS is able to perform

these executions in parallel if the server has multiple processors or support for CPU threading.

3.3.1. *Choosing Synchronization Delays*

There is a tradeoff in TSS between how many states are used (and the synchronization delays for each of them) and the number of commands that must be re-executed on a rollback. The number of states used also determines the responsiveness of the system. Because an inconsistency is not detected until it is executed in the trailing state, the leading state will have progressed further down an incorrect path than in TimeWarp. The longer TSS waits before detecting the inconsistency, the more the states will have diverged, and the more likely it is that the player will notice the rollback when it occurs.

With a synchronization delay of zero, the leading state would provide the fastest updates, allowing clients to see their own commands immediately (if TSS is run at the client side). However, with a synchronization delay of zero, the leading state would also almost always be incorrect, because commands from other players cannot arrive that quickly, and there will be many rollbacks. If the synchronization delay of the leading state is very large, the number of rollbacks will be reduced, but the game will become much less responsive. In Section 4.2, we examine several heuristics to pick this first synchronization delay in order to reduce rollbacks but still provide adequate responsiveness.

Picking the correct number of states and the synchronization delay for each state is also important to the optimal performance of TSS. In order to provide a large enough window of synchronization, if too few states are used the gaps between states must necessarily be large. This leads to greater delay before an inconsistency is detected, which in turn leads to more drastic and noticeable rollbacks. Conversely, if too many states are used, the memory savings provided by TSS will be eliminated, and the cost of executing commands in each state will grow. Additionally, rollbacks will likely be more expensive since a longer cascading rollback involving more state copies is needed before reaching the leading state.

4. Performance Evaluation

There are two main questions we want to ask when evaluating TSS. First, were we correct in our main design assumption that, for a first person shooter, it is significantly more expensive to take a snapshot of the state than it is to execute commands multiple times? Second,

does TSS perform better than other synchronization mechanisms when using similar memory settings?

To answer these questions we performed two different experiments. In the first experiment, we implemented a mirrored server architecture based on the popular Quake I first person shooter, and used TSS as the synchronization mechanism. This allowed us to determine the costs of rollbacks and command executions empirically. To answer the second question, we turned to a simulation of just the synchronization mechanisms, and looked at the performance under different parameterizations. Section 4.1 presents the first experiment, and Section 4.2 the second.

4.1. Command Execution and Rollback Costs

We implemented TSS in the open-source QuakeForge [16] server for the Quake I first-person shooter. For the experiments reported here we set up two mirrored servers, with three clients connected to each (see Fig. 1.c). The servers run TSS to synchronize the commands from the six clients. There is a very low-latency connection between clients and servers. The latency of the server-to-server connection is 50 ms. As mentioned above, the main goal of these experiments is to get a concrete example of the costs of rollbacks and re-executions in a first person shooter. A second result of this experiment was insight into some of the challenges in adding a synchronization mechanism to an existing game engine.

4.1.1. *Implementation*
The first step in implementing TSS in QuakeForge was to alter the server so that all game state data was within an execution context, and create functions capable of copying one execution context onto another. Although for performance reasons Quake uses almost no dynamic memory, within the statically allocated structures the developers of Quake use numerous tricks to avoid extraneous pointer dereferencing. These optimizations make the encapsulation and copying of game state a non-trivial task. In addition, changes had to be made to the game to account for the use of random numbers for events such as item placement. Ideally, a separate pseudo-random number generator (RNG) with the same initial seed would be used to provide consistent random numbers in each execution context. On rollbacks, the RNG state can be included in the copied state in case the leading state had used more random numbers. Unfortunately, Quake was not designed this way and preserving random events turned out to be the major difficulty in implementing and evaluating TSS in QuakeForge.

Table I. Trailing State Synchronization Execution Times in QuakeForge

Run	Synch. Delays (ms)	Execution Time	Executed Commands	**Command Cost**	Rollback Time	Rollbacks	**Rollback Cost**
1	0,50	6.14 s	40,780	**0.15 ms**	1.15 s	817	**1.41 ms**
2	0,100	6.37 s	45,401	**0.14 ms**	1.23 s	870	**1.41 ms**
3	0,50,100	9.02 s	59,981	**0.15 ms**	1.32 s	938	**1.40 ms**
4	0,50,100,150	12.15 s	79,357	**0.15 ms**	1.53 s	1,092	**1.41 ms**
5	0,50,100,500	13.26 s	99,730	**0.13 ms**	3.36 s	2,370	**1.42 ms**

Once the QuakeForge server had been altered to use execution contexts, it was fairly simple to add synchronization. When client commands are received by a mirror, instead of being executed immediately, they are diverted and sent out over the multicast channel to all the mirrors (including the mirror that received the command initially). Upon receiving a multicast command, it is inserted back into Quake's network buffer at the mirror and parsed by the Quake engine. If the command is one that requires synchronization (some commands that do not alter the game state can be executed immediately), it is placed in TSS to be executed later. In the main event loop, the mirror checks each of the TSS states for any pending commands that are ready to be executed. Inconsistencies between the executing trailing state and its leading state are checked for, after commands are executed. In addition to mirroring and synchronization, we also added a trace feature to the QuakeForge server. This logs to a file every command sent to the multicast channel and allows games to be replayed exactly in the future for deterministic simulations.

4.1.2. *Execution Results*
To test the performance of TSS in this mirrored server, we ran a series of simulations using the aforementioned trace feature with different network and synchronization parameters. Unfortunately, due to difficulties accounting for all random events, inconsistencies often occurred, despite the use of a synchronization mechanism. Nevertheless, we were able to gather the cost of rollbacks in Quake. The results in Table I show five runs, all using the same trace file, with three users connected to each of the two mirrors. The statistics were gathered at mirror one, which saw 18,593 commands in total. The "Executed Commands" column shows the total number of Quake moves executed by the mirror, including the $18,593 \cdot \#states$ commands that would be executed with no rollbacks, plus any additional commands re-executed due to rollbacks. The "Execution Time" column shows the total time (system and user) spent executing these commands, measured by the instrumented server. The

"Rollbacks" column shows the number of rollbacks that occurred during the simulation, and the "Rollback Time" column shows the system and user time spent performing the execution context copy, repairing data structures within the execution context, and moving events back to the event queue to be re-executed. "Total Time" includes command execution, rollback, and other functions of the server such as providing reliable multicast. Finally, "Command Cost" and "Rollback Cost" are calculated by dividing the execution time and rollback time by the number of commands and rollbacks respectively.

The command cost is dominated by the actual execution of commands in the Quake engine. TSS event queue management and other bookkeeping turn out to be minor components of the time. Similarly, the rollback cost is dominated by the time to copy the execution context and repair data structures, while the time spent moving executed commands back to the event queue for re-execution is minor. In all of the runs these costs were nearly identical, with command execution being an order of magnitude less expensive than rollback. This confirms our assumption that command executions are far cheaper than state snapshots.

4.2. TimeWarp and TSS Performance Comparison

In order to explore the parameter space for TSS, as well as to compare its performance to that of TimeWarp, we use trace driven simulation of the protocols to collect statistics on a number of metrics relevant to both server and client performance. We do not include MiMaze's bucket synchronization in this comparison because it cannot provide adequate consistency without an unnecessarily large synchronization delay. This makes it unsuitable for a first person shooter such as Quake.

We made several straightforward modifications to TimeWarp to allow a side-by-side comparison of the two algorithms, as follows. To prevent unnecessary rollbacks from local commands being executed immediately, a synchronization delay d is added to TimeWarp. To reduce and bound memory usage, we also provide a maximum time l for which consistency must be maintained and a fixed number of snapshots s TimeWarp can use. The time between snapshots is computed as $(l - d)/s$ (we generate a new snapshot based on time since last snapshot, not the number of commands since last snapshot). As in TSS, we assume that TimeWarp will not issue any anti-messages on rollback.

4.2.1. *Trace Collection*

Ideally, we would like to simulate these two mechanisms using traces gathered at each of the nodes of a mirrored or distributed FPS game deployed on the Internet. These traces would contain the time at which each command is sent by its issuing mirror, and the time at which the command arrives at every other mirror. Unfortunately, for a number of reasons, collecting these ideal traces is not practical. First and foremost, all popular first-person shooters presently use the client-server architecture. To solve this problem we constructed artificial mirrored architecture traces from available game server logs.

To gather traces of typical game play, we instrumented and ran a well-connected Quake III server at the University of Michigan. We collected the round trip times (RTTs) between the server and each connected client once per second over several periods in January, 2002. Quake III was chosen because it is far more popular than QuakeForge, making it easier to get traces with more than one or two active players. The game itself and its network properties are roughly the same as Quake I or any of the other first person shooters for which TSS was designed.

We obtained client RTTs from two sources: the Quake III server's internal RTT measurements and an external ping program. The server was modified to record its internal RTT measurements to each client in the game log once a second. The external program sends a ping to each client once per second as long as they are connected to the server. It relies on TCP pings [5] as opposed to ICMP pings, since the latter are more often blocked by ISPs or personal firewalls. A comparison of the server and external ping data shows that the server measurements contain a significant amount of application-level latency, as well as additional jitter. Therefore, our simulations rely on the more accurate external ping data. These latencies are between the clients and the server, not between pairs of clients as it would be in a mirrored architecture. True client-to-client latency measurements would have required cooperation from every player who used our server, asking them to run measurements and return the results to us. Many players are very untrusting (fearing viruses or trojan horses), and we did not feel this was a realistic goal. Instead, we assume that client RTT is nearly symmetric, and therefore the one-way delay between our server and a client is half the measured ping time. We then approximate the end-to-end latency between two clients to be the sum of the one-way delays between the clients and the server. The consequence of this is that the topology is somewhat artificial, *but still provides a suitable basis for comparing the different protocols in identical conditions.* To approximate the origination time of a packet from a mirror, we use the

Table II. Log Snippets Used

Number of Mirrors	Total Moves	Gametime (mm:ss)
2	49,590	18:47
4	94,195	15:55
6	48,684	05:30

time the packet arrives at our server minus the most recently observed one-way delay. The arrival times at the other mirrors is calculated as packet origination time plus the approximated one-way delay in the star topology.

The timestamps within the Quake III server log only have the one second granularity provided by the `localtime()` function in the standard C library. Because of this, the log does not contain exact arrival times of the 30-50 commands per second from each client which we need for our simulations. To remedy this lack of information, the command inter-arrival times for the clients are generated by a model derived from `tcpdump` [11] logs of Quake III games. We used the Quake III client to connect to several Quake III servers on the Internet and recorded the game traffic locally with `tcpdump`. We then used Ethereal [6], which has a built-in packet decoder for Quake III, to filter out everything except the game commands sent out by our client. We computed the cumulative distribution function of the gaps between commands, and used this distribution to randomly pick inter-arrival times for the commands in the trace. With this last step we now have a trace which contains the command issue and receive times for each of the mirrors, as we wanted in the ideal case.

A simplification we make for our simulations is to break the traces up into smaller *snippets* of gameplay. The set of clients is constant for the duration of the snippet, and the game does not reset or change levels. We do not look at dynamic adjustment of any of the synchronization mechanisms below, so including clients joining and leaving would merely complicate the presentation of the results without gaining any additional insight For our simulations, we used a variety of snippets, focusing on those with longer playtime. We look at different numbers of mirrors (clients) and different distributions of mirror-to-mirror latencies when selecting the snippets to use.

Below, we examine three different snippets in our simulations. Table II shows the number of moves and total time elapsed for the three
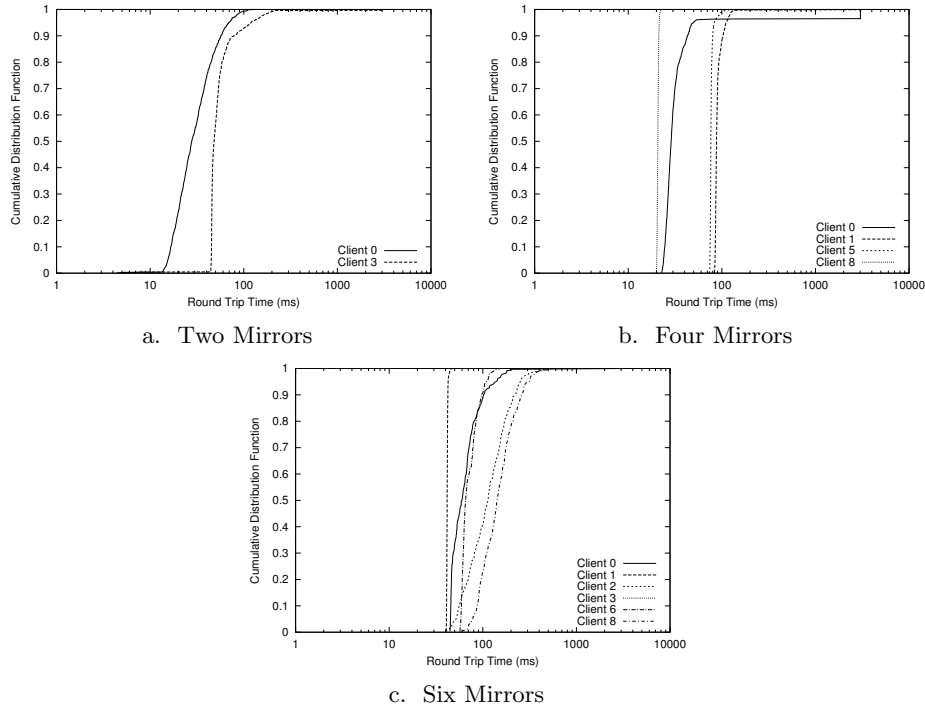
a. Two Mirrors

b. Four Mirrors

c. Six Mirrors

Figure 6. Snippet Ping Time Distributions.

snippets. Fig. 6 shows cumulative distribution of the ping times to each of the clients in these traces.

4.2.2. *Simulation*

There are several metrics of interest when trying to compare the performance of these two synchronization methods. First, there is the total number of rollbacks needed. This is important both to the server, where rollbacks are expensive and limit the server's performance, and to the client, where rollbacks can cause unexpected changes to the view of the world and disrupt gameplay. All rollbacks are not equal however, so we also look at the rollback magnitude (which we abbreviate $\mathcal{M}$ in the results). The magnitude of a rollback is how long the state has been incorrect due to the inconsistency. Rollback magnitude can be measured either in time (difference between when the inconsistency was caught and when the command should have been executed) or in executed commands (number of commands executed after the inconsistent one). Figures 2.a and 5.b show how magnitude is computed for TimeWarp and TSS respectively. In the simulation results below, we measure magnitude in terms of time. Since Quake has regular packet rates, this

can easily be converted to an approximate number of commands. The higher the magnitude of a rollback, the more things that could have diverged between the correct state and the executing state. This means that when the rollback occurs, high magnitude rollbacks are more likely to be noticed by the client.

From the server's perspective, another important metric is the amount of work that must be done during a rollback. This includes re-executing the commands after the inconsistency in order to return to the correct execution time. In TimeWarp, when the snapshot used to recover from is older than the point where the inconsistency occurs, the commands between the snapshot and the inconsistency must also be re-executed. This metric, combined with the total number of rollbacks, determines the cost of the synchronization mechanism on the server.

There are a number of questions we hope to answer with these simulations. The primary goal is to determine which synchronization mechanism performs best when used in a first person shooter with a high packet rate. We are interested to find out which synchronization mechanism requires the least server resources and which disrupts the clients' gameplay the least. A second related question is how TSS and TimeWarp compare in a scenario where there is a lower packet rate, a situation disadvantageous to TSS since there are fewer potential commands to amortize rollbacks. In addition to head-to-head comparison of the two protocols, we would also like to know what role the different parameters of TSS and TimeWarp play in overall performance. Do certain heuristics for picking the synchronization delays result in fewer rollbacks or a decrease in rollback magnitude? Do they do so consistently for different sets of mirrors?

To perform this comparative study of TSS and TimeWarp, we created two simulators that receive as input the trace snippets and produce as output results for the metrics described above. The simulators simulate only synchronization related operations of their respective mechanisms, not the actual gameplay. The TimeWarp simulator allows us to modify the frequency of snapshots and the point at which snapshots are thrown away. There is also a parameter that adds a synchronization delay between the origination time of the packet and when each of the mirrors schedule execution of the command. The TSS simulator allows us to specify the number of states to use and the synchronization delay for each of the states.

When comparing the two synchronization mechanisms, we tried to configure them to have similar memory usage, consistency, and added minimum delay. Therefore we used n states in TSS and $n-1$ snapshots (plus the additional actual game state) in TimeWarp. The same synchronization delay was used for TimeWarp and the first state in TSS.

Table III. Snippet Heuristics Based on Delay Observations

Number of Mirrors	Synchronization Delay (ms)		
	Maximum Mean	Maximum 90%-tile	Aggregate 90%-tile
2	51	68	66
4	117	91	89
6	143	243	177

Similarly, the maximum time used for TimeWarp is the synchronization delay used for the last state in TSS. These settings fully configure Time-Warp, but with TSS there is also the issue of assigning synchronization delays to the remaining state(s). We look at how this is done below.

As mentioned above, one of the questions we would like to answer is what impact the parameters have on performance. In particular, we look at how setting the initial synchronization delay affects the performance of each of the algorithms. We look at different heuristics that analyze the entire snippet and produce a synchronization delay to be used when executing that snippet. Ideally, when selecting the first synchronization delay we want something low enough that it does not hurt gameplay (by adding too much additional latency), while at the same time we want it high enough that most commands do not generate rollbacks. If latencies were constant, this would simply be the maximum of the end-to-end latencies to each of the other mirrors. A lower synchronization delay would cause moves from one client to always arrive late and cause a rollback; a higher one will unnecessary add additional delay to each command.

The first heuristic we investigate is to set the synchronization delay to the maximum over all pairs of mirrors of the mean end-to-end delay between the two mirrors. We call this heuristic "Maximum Mean". The conjecture here is that in a controlled environment where all the mirror-to-mirror latencies are roughly the same and variance is low (e.g., on the private internal network of the mirrored server architecture), the maximum of the means would be an aggressive setting, having a low added delay; at the same time, it would not cause many rollbacks and the rollbacks caused would have low magnitudes since the delay variance is low. In more heterogeneous environments, this heuristic would likely perform much worse since it would tend to be skewed by outliers.

Another heuristic we consider is to set the synchronization delay to the maximum of the 90%-tile end-to-end delays between each pair of mirrors, called "Maximum 90%-tile". The intent of this heuristic is to conservatively handle wide variances in delay, where a few very late commands would skew the mean in the first heuristic. By choosing the 90%-tile of the maximum, our hope is that rollbacks occur only on network anomalies, while wider delay distributions with many packets arriving after the mean do not suffer from constant rollbacks.

The final heuristic we look at is setting the synchronization delay to the 90%-tile of all end-to-end measurements, which we call "Aggregate 90%-tile". This heuristic would help ensure that few total rollbacks would occur overall, akin to the Maximum 90%-tile heuristic. By using the aggregate delay observations, we more aggressively seek to generate rollbacks only for outliers. However, this heuristic would be disadvantageous if there is a single high delay mirror in an environment with few delay outliers. In this case, because the delay is aggregated, the mirror with high delay would have most of its commands seen as outliers, and there would be a high likelihood that that mirror's commands generate rollbacks.

For each of the six snippets (three from each trace), we use these heuristics to compute synchronization delays. The synchronization delays calculated and used in our simulations are shown in Table III. We will evaluate the effectiveness of these heuristics in Section 4.2.5

The second parameter to the simulators, the final synchronization delay/maximum time, we set to a fairly conservative two seconds. This was large enough so that all but a few late commands were able to be recovered from, but not so large that performance suffers due to it. TimeWarp is particularly vulnerable to this parameter being too large, since it must evenly space its snapshots between this value and the initial synchronization delay.

Our choice of number of states and the delay for each state was based on the observation that we wanted rollbacks to cover as little time as possible, but we also wanted to limit the memory and CPU overhead of the synchronization mechanism. In the experiments below, we use three states for each synchronizer: the main state and two snapshots for TimeWarp and three states for TSS. For TSS, the first and last synchronization delays are set as described before, and the middle synchronization delay is set to be double the minimum synchronization delay.

4.2.3. *Simulation Results*

The simulators were executed with each of the three snippets, using the parameters specified by each of the heuristics for a total of nine

Table IV. Trace-Based Simulation Results

a. Maximum Mean

Number of Mirrors	Sync. Alg.	Rollbacks	Re-execs	$\mathcal{M}$ (ms)
2	TW	11,377	299,429	43
2	TSS	7,737	24,568	62
4	TW	5,511	852,121	1,358
4	TSS	455	66,296	1,766
6	TW	26,440	2,089,045	55
6	TSS	5,936	121,704	153

b. Maximum 90%-tile

Number of Mirrors	Sync. Alg.	Rollbacks	Re-execs	$\mathcal{M}$ (ms)
2	TW	4,233	119,515	86
2	TSS	2,497	10,046	84
4	TW	9,819	1,069,622	919
4	TSS	2,136	82,406	932
6	TW	2,462	237,686	241
6	TSS	324	15,037	334

c. Aggregate 90%-tile

Number of Mirrors	Sync. Alg.	Rollbacks	Re-execs	$\mathcal{M}$ (ms)
2	TW	4,955	137,679	75
2	TSS	3,013	11,482	80
4	TW	11,845	1,170,099	868
4	TSS	2,977	89,707	905
6	TW	10,599	877,512	90
6	TSS	2,066	54,839	193

side-by-side comparisons for each trace. The results of the simulations are presented in Table 4.2.2 for the realistic trace and in Table V for the unrealistic worst-case trace. We look at (1) the total number of rollbacks, (2) the total number of re-executed commands, and (3) the average magnitude ($\mathcal{M}$) of rollbacks. The results for each snippet are totals for all of the mirrors in that snippet combined. From Table 4.2.2, we see that, due to its ability to repair bursts of late commands with a single rollback, TSS has fewer rollbacks than TimeWarp. In most cases,

the number of rollbacks with TSS is dramatically lower, but even in its worst cases TSS has 40% fewer rollbacks than TimeWarp.

Fewer commands are re-executed by TSS than by TimeWarp due to rollbacks. Consider the results for two mirrors. In this case, the number of rollbacks is similar, but the re-executed commands are an order of magnitude lower for TSS. The reason for this difference is that TimeWarp must roll back from fixed interval snapshots. For this set of mirrors, the rollback magnitudes are very small (indicating there are few commands after the inconsistency that must be re-executed), but they often have to be created from a relatively old snapshot. All commands in the interval between when the snapshot was taken and the point where the inconsistency occurred must be re-executed. In TSS, the trailing state used in the rollback is always at the point where the inconsistency occurred in the leading state, so there is no need to re-execute commands to catch up from an outdated snapshot.

Not shown in Table 4.2.2 is the cost of initial execution of commands. It must be kept in mind that TSS executes every command once for each state, so in these examples there are three times as many normal executions in TSS as in TimeWarp. Similarly, TimeWarp must make snapshots approximately once every second while TSS requires no state copying unless a rollback occurs. This cost is not accounted for in Table 4.2.2 either. These costs are constant, so the server can be properly provisioned to handle them. Re-execution costs, however, happen unpredictably, and may become very expensive. We consider these to be the larger problem. Even if rollbacks are rare, if they are so expensive that the game must stop for a second while the rollback is performed, gameplay will suffer noticeably. Even with the cost of initial execution and snapshots figured in, TSS places a lower load on the server than TimeWarp.

The average magnitude of rollbacks is, as expected, higher for TSS than for TimeWarp, since TSS waits before generating rollbacks while TimeWarp always rolls back immediately. Depending on the snippet and heuristic, the difference in magnitudes ranges from roughly even in the 2 mirror case to nearly three times greater for TSS in the 6 mirror case. Because we set the second synchronization delay in TSS to be twice the initial synchronization delay, this limits the minimum possible magnitude of a rollback in TSS. Especially in the 6 mirror snippet, where the synchronization delays are large, this leads to a noticeable difference between TSS and TimeWarp. Better tuning of the middle synchronization delay(s) in TSS could reduce this magnitude.

When considering the impact of the average magnitude of rollbacks, it is important to remember that it is the average for the rollbacks *that actually occurred.* So although the average rollback in TSS may be

Table V. TSS versus TW using Server Aggregated Moves and Maximum 90%-tile Heuristic.

# Mirrors	Sync. Alg.	Rollbacks	Re-execs	$\mathcal{M}$ (ms)
2	TW	157	138	17
2	TSS	156	217	68
4	TW	408	1,845	930
4	TSS	322	1,745	1,181
6	TW	112	357	201
6	TSS	82	430	353

more noticeable, there are so many more rollbacks in TimeWarp that considering the total magnitude (Rollbacks·$\mathcal{M}$), TSS again outperforms TimeWarp for all nine cases. Less encouraging however are the rollback magnitudes themselves for both TSS and TimeWarp. Especially for the 4 mirror snippet, where Mirror 0 has a heavy tail in its delay distribution, very large rollbacks that are likely to be noticeable to the players do occur. Without a significantly larger initial synchronization delay, there is nothing any synchronization algorithm can do to improve this situation.

4.2.4. *Worst-case TSS Performance*
We also looked at a second trace where the mirrors generated only one move per second. It is highly unrealistic to assume a move issuance rate of one move per second in any fast-pace FPS. We use present these results here to show that even in this worst-case scenario for which TSS was not designed, it does not fall apart. Table V presents the results for the trace containing only one move per second from each mirror. We would expect TSS to perform poorly in this situation, since it loses the ability to combine many rollbacks into one, while it still pays the penalty in rollback magnitude by waiting to execute the rollback. We only show the Maximum 90%-tile heuristic case, but the results for the other two heuristics are comparable.

Since both TSS and TimeWarp have nearly the same numbers of rollbacks, by total rollback magnitude TimeWarp holds an edge for client performance in this unrealistic scenario. Nevertheless, TSS still has fewer total rollbacks than TimeWarp with this trace, which can be attributed to commands from different mirrors being late at the same time, allowing TSS to occasionally repair multiple inconsistencies. The number of commands re-executed are roughly the same for both TSS and TimeWarp, since with fewer commands, an outdated snapshot is

no longer as damaging to TimeWarp. Once again, TSS outperforms TimeWarp with regards to server performance, although the difference is much smaller in this case. Finally, the rollback magnitude is higher for TSS, but is less than twice as expensive for the four and six mirror cases, and still below 100 ms for the two mirror case.

4.2.5. *Heuristics for Setting Synchronization Delay*
Looking back at Table 4.2.2, there is no clear winner among the three heuristics used to compute the synchronization delay that performs best for all three snippets, although Maximum 90%-tile has a slight edge. Maximum Mean has more rollbacks and re-executions than the other two heuristics in the 2 and 6 mirror cases because in these cases the mirrors have a high percentage of their delays above the mean. The average magnitude of these rollbacks however is the lowest when using Maximum Mean, since the delay before detecting inconsistencies is lower than with the other heuristics. In the 4 mirror case, Maximum Mean is skewed by the outliers in Mirror 0, and produces a delay which is too conservative. The result is few rollbacks, but with a very high rollback magnitude. Aggregate and Maximum 90%-tile perform about the same in the two and four mirror cases with similar rollback magnitudes and numbers of rollbacks, with more differentiation between the two in the six mirror cases. Aggregate 90%-tile has lower rollback magnitudes, while Maximum 90%-tile has fewer rollbacks and re-executions. Turning to the Rollbacks $\cdot \mathcal{M}$ total magnitude again, we give the edge to Maximum 90%-tile since although its rollbacks have a higher magnitude there are few enough of them to make up for it.

5. Conclusion

In this paper we have presented TSS, an optimistic synchronization mechanism designed for multiplayer games with low latency but strong consistency requirements. It provides for low-latency, consistent gameplay through the use of multiple copies of the game state and rollbacks. We have shown that because of its design characteristics, TSS performs well in high-speed games where there is a large game state and many commands to be synchronized. There are still a number of areas where future work is needed. The benefits of event classification have not been explored in depth. This has the potential to greatly reduce the number of rollbacks experienced in TSS. In the mirrored server architecture (where the mirrors can be trusted), there is potential for performing some pre-execution at one mirror to reduce the cost of executing in each state at each mirror. Similarly, in purely peer-to-peer environ-

ments there is the ability to employ interest management. Neither of these optimizations has been explored yet. More accurate game traces, such as those recorded by `pkthisto` [1], could be used for simulating TSS and other synchronization algorithms, although the traces we have used in this paper provide an adequate means of comparing TSS and TimeWarp side by side. The problem of selecting the right leading synchronization delay was examined, but the selection of intermediate synchronization delays or the dynamic adjustment of synchronization delays as network conditions change is still unexplored.

References

1. Armitage, G.: 2001, 'pkthisto-0.1.2'.
   http://opax.swin.edu.au/∼garmitage/q3/quake-server-mods.html.
2. Baughman, N. and B. Levine: 2001, 'Cheat-Proof Playout for Centralized and Distributed Online Games'. In: *Proc. Infocom 2001*.
3. Bernier, Y.: 2001, 'Latency Compensating Methods in Client/Server In-Game Protocol Design and Optimization'. In: *Proc. of GDC 2001*.
4. Cronin, E., B. Filstrup, and A. Kurc: 2001, 'A Distributed Multiplayer Game Server System'. UM EECS589 Course Project Report,
   http://www.eecs.umich.edu/∼bfilstru/quakefinal.pdf.
5. Dykes, S., K. Robbins, and C. Jeffery: 2000, 'An Empirical Evaluation of Client-side Server Selection Algorithms'. In: *Proc. of IEEE Infocom 2000*.
6. et al., G. C., 'The Ethereal Network Analyzer'. http://www.ethereal.com/.
7. Francis, P., S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang: 2001, 'IDMaps: A Global Internet Host Distance Estimation Service'. *IEEE/ACM Transactions on Networking* **9**(5), 525–540.
8. Gautier, L., C. Diot, and J. Kurose: 1999, 'End-to-end Transmission Control Mechanisms for Multiparty Interactive Applications on the Internet'. In: *Proc. of IEEE Infocom 1999*, Vol. 3.
9. Helder, D. A. and S. Jamin: 2002, 'End-host Multicast Communication Using Switch-tree Protocols'. In: *Proc. of GP2PC*.
10. id Software, 'Quake'. http://www.idsoftware.com/.
11. Laboratory, L. B. N., '`tcpdump`'. http://ee.lbl.gov/.
12. Lamport, L.: 1978, 'Time, Clocks, and the Ordering of Events in a Distributed System'. *Communications of the ACM* **21**(7), 558–565.
13. Lincroft, P.: 1999, 'The Internet Sucks: Or, What I Learned Coding *X-Wing vs. TIE Fighter*'. In: *Proc. of Game Developers Conference 1999*.
14. Mauve, M.: 2000, 'How to Keep a Dead Man from Shooting'. In: *Proc. of the 7th International Workshop on Interactive Distributed Multimedia Systems*. pp. 199–204.
15. Mauve, M., S. Fischer, and J. Widmer: 2002, 'A Generic Proxy System for Networked Computer Games'. In: *Proc. of NetGames2002*.
16. Project, T. Q., 'QuakeForge'. http://www.quakeforge.net/.
17. Steinman, J. S.: 1995, 'Scalable Parallel And Distributed Military Simulations Using the SPEEDES Framework'. In: *ELECSIM95*.

18. Steinman, J. S., R. Bagrodia, and D. Jefferson: 1993, 'Breathing Time Warp'. In: *Proc. of the 1993 Workshop on Parallel and Distributed Simulation*. pp. 109–118.
19. Steinman, J. S., J. W. Wallace, D. Davani, and D. Elizandro: 1998, 'Scalable distributed military simulations using the SPEEDES object-oriented simulation framework'. In: *Proc. of Object-Oriented Simulation Conference (OOS'98)*. pp. 3–23.