An Efficient Synchronization Mechanism for Mirrored Game Architectures

Eric Cronin   Burton Filstrup   Anthony R. Kurc   Sugih Jamin[*]
Electrical Engineering and Computer Science Department
University of Michigan
Ann Arbor, MI 48109-2122
{ecronin,bfilstru,tkurc,jamin}@eecs.umich.edu

ABSTRACT

Existing online multiplayer games typically use a client-server model, which introduces a single bottleneck and point of failure to the game. Distributed multiplayer games 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.

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 clients in the form of update messages. This adds additional latency over the minimum cost of sending commands directly to other clients (ignoring possibilities such as asymmetric routing). 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, due to network delay and other factors, clients' game states would diverge over time, leading to inconsistencies.

Common synchronization techniques used in existing distributed simulation environments include bucket synchronization, breathing bucket synchronization, and Time Warp synchronization [5, 13]. Many of these synchronization mechanisms were initially designed for use in large-scale military simulations, and then were 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 with distributed first-person shooter games such as Quake [7] 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 either introduce too much additional latency or provide only loosely consistent synchronization. TSS is designed to execute commands as quickly as possible while at the same time maintaining a consistent copy of the game state at all players.

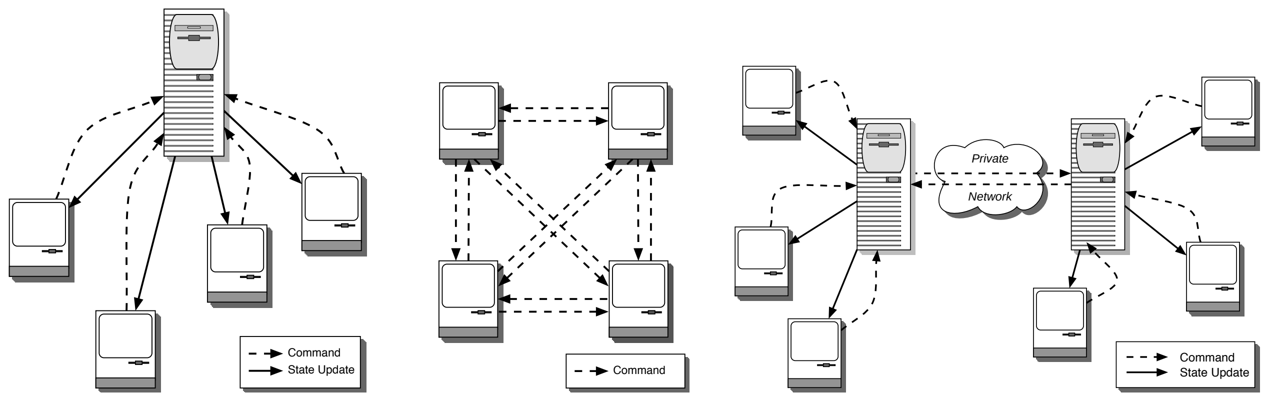
The remainder of the paper is organized as follows: in Section 2 we provide background on multiplayer game architectures, including the mirrored game architecture, and previously proposed synchronization methods. In Section 3 we introduce trailing state synchronization. Section 4 shows some preliminary performance figures on TSS and finally Section 5 concludes.

2. BACKGROUND

2.1 Synchronization Techniques

In peer-to-peer games, instead of sending commands to a central server which 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

[*]This project is funded in part by NSF CAREER Award ANI-9734145, the Presidential Early Career Award for Scientists and Engineers (PECASE) 1998, and the Alfred P. Sloan Foundation Research Fellowship 2001. Additional funding is provided by equipment grants from Sun Microsystems Inc. and Compaq Corp.



(a) Client-Server **(b) Peer-to-Peer** **(c) Mirrored Server**

Figure 1: Multiplayer Game Architectures.

state, there needs to be some mechanism to guarantee a global ordering of events [8]. 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. 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 delay, no one will be willing to play the game.

2.1.1 *Conservative Algorithms*

Lockstep synchronization [4], used in military simulations, is by far the simplest technique 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 and wall-clock time. There is no way to guarantee that the game will advance at a regular rate, much less at a fast enough rate for interactive gameplay. In a multi-player game, this is not an acceptable tradeoff. There are a number of similar algorithms which perform better than lockstep but are still unable to maintain a constant rate in all situations. One of these is fixed time-bucket synchronization, where a synchronization delay is used to reduce the dependency on latency between members. An optimistic version of this algorithm is discussed below. For the reasons mentioned above, these "conservative" algorithms perform poorly in fast-paced games where a constant rate of simulation is important, although they are still suitable for slower turn-based games.

2.1.2 *Optimistic Algorithms*

The second approach to ensuring consistency is to detect and correct any differences. These algorithms execute events optimistically 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 for use with Quake before describing TSS.

Time Warp synchronization [13] works by taking a snapshot of the state at each execution, and issuing a *rollback* to an earlier state if an event earlier than the last executed event is ever received. On a rollback, the state is first restored to that of the snapshot, and then all events between the snapshot and the execution time are re-executed. Periodically, all members reset the oldest time at which an event can be outstanding, thereby limiting the number of snapshots needed. One complication with rollbacks is that Time Warp assumes that events directly generate new events. As part of the rollback, anti-messages are sent out to cancel previously generated events that have become invalid (which in turn 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.

Another limiting feature of Time Warp synchronization in a game such as Quake is the requirement to checkpoint at every message. A Quake 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 higher frame rates and larger contexts). Additionally, copying a 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 [10], 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.

"Breathing" algorithms [12] attempt to solve the problem of excessive rollbacks seen in Time Warp by restricting the amount of execution that can be done optimistically. Instead of fully optimistic execution, breathing algorithms limit their optimism to events within an *event hori-*

zon. Events beyond the horizon can not be guaranteed to be consistent, and are therefore not executed. 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 (which may be influenced by earlier events), and it is therefore not clear how to accurately define an event horizon.

The algorithm implemented in MiMaze [5] is an optimistic version of the conservative bucket synchronization algorithm. 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 at a particular bucket, the previous bucket's event is *dead reckoned*, for example by replaying the previous command; if multiple events are available in a 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 per bucket is executed. For a simple game such as MiMaze, these optimizations at the cost of consistency may be acceptable. 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. With a game like Quake though, 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. TSS uncouples these two parameters, allowing for better control.

2.2 Mirrored Game Servers

Despite the added latency, single point of failure, and scalability problems in the client-server architectures mentioned in the introduction, they are by far the most commonly used architecture in current multiplayer games. We believe there are a number of reasons behind the popularity of client-server architectures. First, the networking code is simpler 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 and copy protection and to easily update clients. 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 required [9]. End-host multicast [6] 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 [2] is a hybrid architecture designed to address the problems with client-server and peer-to-peer architectures. 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 [3] 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 used.

Since there are now multiple servers for the same 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, 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 clients of a peer-to-peer architecture is important because cheating can become easier in distributed architectures [1].

The use of mirrored servers does place some restrictions on the synchronization algorithm used. Each mirror must be able to handle multiple clients which could be located in any part of the game's world at any given time. As a consequence, it is not straitforward 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 with each other, reducing the consistency requirements. It is, however, still possible for mirrors to do interest management to individual clients as is common in client-server architectures, reducing the bandwidth required between mirror and client and preventing clients from knowing information they do not absolutely need (reducing the opportunities to cheat).

3. TRAILING STATE SYNCHRONIZATION

As described above, none of the existing distributed game or military simulation synchronization algorithms are entirely suited to a game such as Quake which has very frequent updates, has a need for strong consistency, and is very latency sensitive. Our solution to this problem is trailing state synchronization (TSS). Similar to Time Warp, TSS is an optimistic algorithm, and must execute rollbacks when inconsistencies are detected. However, it does not suffer from the high memory and processor overheads of Time Warp.

When rollbacks are required, instead of copying the state from a snapshot taken just prior to the offending command as Time Warp does, TSS copies the state from a second copy of the same game which 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 is to be repaired (in fact, it is this second state which detects that an inconsistency has occurred).

Instead of keeping snapshots at every command (or every few commands as in [10]), TSS keeps multiple copies of the same game state, each at a different simulation time. These

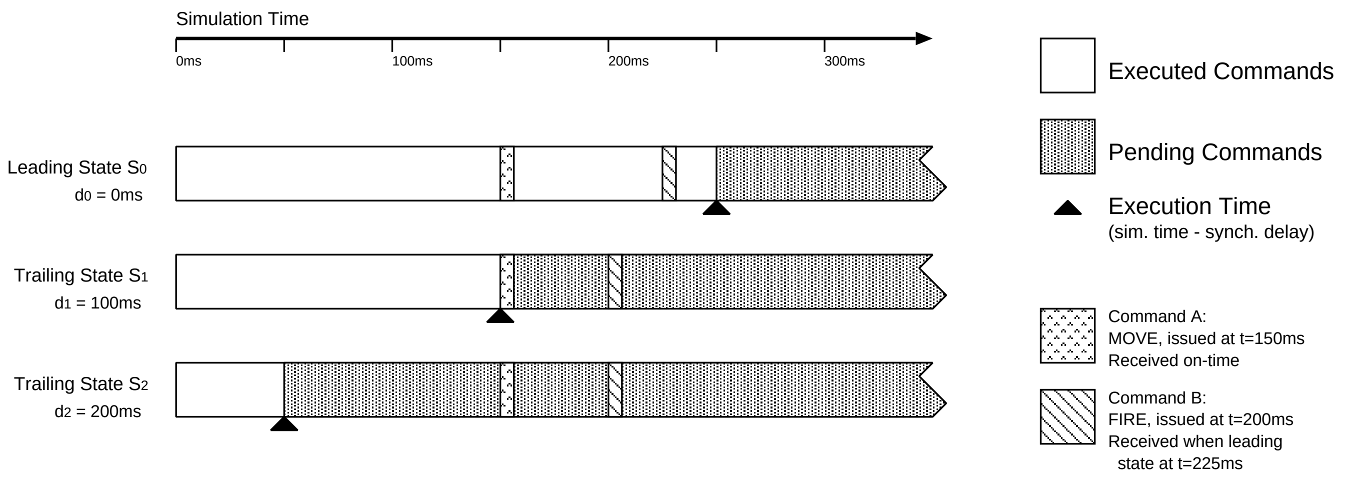


Figure 2: Trailing State Synchronization Execution.

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 screen, 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.

TSS is able to provide consistency because each trailing state will see fewer misordered commands than the state preceding it by waiting longer for delayed commands to arrive before executing. The leading state executes with little or no synchronization delay (d_0). The synchronization delay for a state is defined as the difference between the current *simulation time* (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 in each state 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. With a synchronization delay of zero, the leading state will provide the fastest updates, allowing clients to see their own commands immediately and preserving the "twitch" nature of the game. However, with a synchronization delay of zero, the leading state will also frequently be incorrect in its execution, which later states will have to detect and correct.

Informal usability tests performed during the evaluation of TSS show that rollbacks of less than 100 ms are unnoticeable to most players. 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 the notification that the player has been killed slightly in case there is a rollback. These problems are no worse in TSS than in any other dead reckoned game which repairs inconsistencies.

As commands arrive (or are generated) at a client, they are placed on a pending list for each trailing state. Late moves whose timestamps are earlier than the current execution time for a state are placed at the head of the event list for a state and executed immediately. The different copies of a command in each trailing state are linked together so that the command from state S_n can find the copy of the same command that was executed in the preceding state S_{n-1}. In order 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 at the directly preceding state. 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 there will go undetected. However, if it is assumed that the longest synchronization delay, as in the other bounded optimistic algorithms, is large compared to expected command transit delays, this is unlikely to pose a problem.

In Quake, there are two basic classes of events that a command can generate. The first type we refer to as *weakly consistent*, and consists of move commands. With these events, 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 is *strictly consistent*, and for these events, such as weapons being fired (particularly projectiles), it is important that both states agree on exactly when and where the event occurred.

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 which were executed in the incorrect state after the rollback time. The next time the leading state executes, it will re-execute the commands and return to the proper execution time. If a rollback occurs between states S_n and S_{n-1}, it is possible that once S_{n-1} re-executes to correct the inconsistency, new inconsistencies between S_{n-1} and S_{n-2} will be found. In this fashion, any inconsistencies in a trailing state that

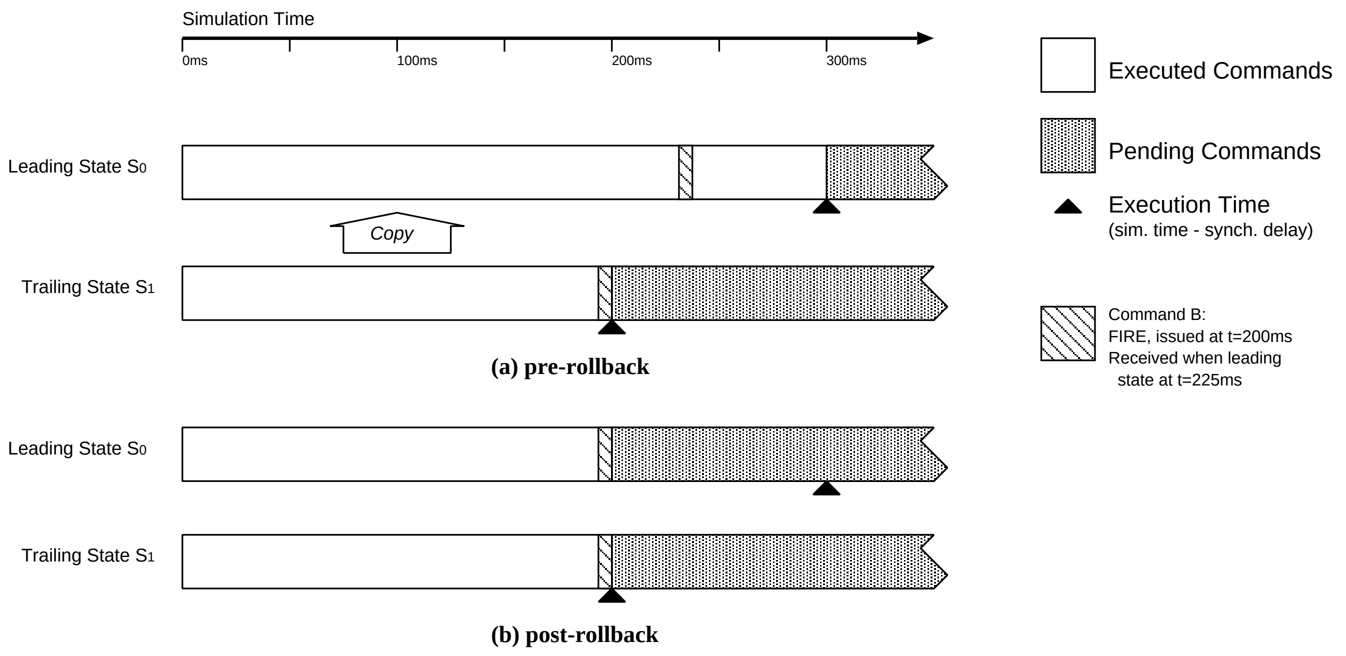


Figure 3: Trailing State Synchronization Rollback

the leading state also shares will be corrected. The problem of anti-messages present in Time Warp is avoided in TSS because new commands are not directly related to any single previous command. Without this direct relationship, there is no straightforward way to create anti-messages. Instead we take the same approach often used in dead reckoned games, applying the new commands to the corrected state.

There is a tradeoff in TSS between how many states are used (and the synchronization delay gap between them) and the number of commands which must be re-executed on a rollback because the inconsistency is not detected until it is executed in the trailing state. However, with weakly consistent events, it is possible that not every late or misordered command will require a rollback, and TSS is able to take advantage of this while other algorithms may not be.

3.1 An Example of TSS

Fig. 2 and Fig. 3 depict a simple example of TSS. There are three states in the example with delays of 0 ms, 100 ms, and 200 ms respectively. We focus on two commands. Command A is a MOVE, issued (locally) at simulation time $t = 150$ and executed immediately in the leading state. At time $t = 250$, the first trailing state reaches an execution time of 150 and executes command A. Since A was on time, its execution matches the leading state's execution and no inconsistency occurs. Similarly, at time $t = 350$, the final trailing state reaches execution time 150 and executes command A. It too finds no inconsistency, and no one is left to check it (this is a contrived example, in real life a longer delay than 200ms on the last trailing state would likely be used) so the command is removed from all three states.

Command B is a FIRE event, issued at simulation time $t = 200$ by a remote client. By the time it arrives, the simulation time locally is $t = 225$. The command is executed immediately in the leading state and placed in the proper position in the two trailing states since they are at execution times 100 and 0 respectively. At time $t = 300$, the first trailing state executes B. When it compares its results with the leading state's results, it is unable to find a FIRE event from the same player at time 200, and signals the need for a rollback. Fig. 3 zooms in on the recovery procedure. The state of the trailing state is copied to the leading state, which places it at execution time 200 (moves up to $t = 200$ have been executed, moves later than $t = 200$ have not). The leading state then marks all commands after time 200 as unexecuted, which is seen in Fig. 3(b). The leading state will then re-execute the commands in the shaded area up to the current execution time, which has not changed during the rollback. This example exposes one of the features of TSS. It is possible that there were other inconsistencies in the gap between times 200 and 300 (a burst of congestion perhaps). The recovery of the first inconsistency at time 200 in effect canceled any future recoveries in this window since all commands between 200 and 300 are re-executed.

3.2 Analysis of TSS

Although similar to many other synchronization algorithms, TSS has key differences with each of them. TSS is clearly very different from any of the conservative algorithms, since its execution is based on when the synchronization delay schedules a move and not when all events for a time period have arrived. It is also clearly different from MiMaze's bucket synchronization since it provides absolute synchronization as long as events are delayed no more than the longest synchronization delay. MiMaze on the other hand does not really detect, let alone recover from, any inconsistencies that may occur. TSS and Time Warp both execute commands as soon as they arrive. They differ however in their methods of recovering from inconsistencies. TSS is a little more optimistic than Time Warp in that it does not keep a snapshot of the state before executing every command so that it can recover as soon as a late command ar-

Run	Synchronization Delays (ms)	Executed Commands	Execution Time (sec)	Rollbacks	Rollback Time (sec)	Total Time (sec)	Command Cost (ms)	Rollback Cost (ms)
1	0,50	40,780	6.14	817	1.15	8.95	0.15	1.41
2	0,100	45,401	6.37	870	1.23	9.32	0.14	1.41
3	0,50,100	59,981	9.02	938	1.32	12.30	0.15	1.40
4	0,100,1000	331,687	26.20	6,687	10.11	43.77	0.08	1.51
5	0,50,100,150	79,357	12.15	1,092	1.53	15.90	0.15	1.41
6	0,50,100,500	99,730	13.26	2,370	3.36	19.38	0.13	1.42
7	0,50,500,1000	251,044	23.22	6,995	10.51	39.27	0.09	1.50

Table 1: Trailing State Synchronization Execution Times

rives. Instead, it catches inconsistencies by detecting when the leading state and the correct state diverge, and correcting at that point. It is possible, especially with weakly consistent events, that though executed out of order, commands may not cause an inconsistency. Other synchronization algorithms, which look only at the command and not the results of the command, are unable to take advantage of this.

TSS performs best in comparison to other synchronization algorithms when three situations are present in the game: the game state is large and expensive to snapshot, the gap between states' delays is small and easy to repair, and event processing is easy. The first is definitely present in Quake and other first-person shooters, with one megabyte or more of data per context. The second is a parameter of TSS and can be tuned (see Section *3.2.1*). The cost of processing each command is not immediately apparent, and must be determined experimentally. We present a preliminary study of the performance of TSS in Section 4. For games like Quake, the results indicate that it is significantly cheaper to execute a command multiple times than to make snapshots of the game state.

3.2.1 Choosing Synchronization Delays

Picking the correct number of states and the synchronization delay for each state is critical to the optimal performance of TSS. In order to provide a large enough window of synchronization the gaps between states must necessarily be large if too few states are used. 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. Additionally, rollbacks will likely be more expensive since a longer cascading rollback is needed before reaching the leading state. Given information on the network and client parameters (delay distribution, message frequency, the cost of executing a message, the cost of a context copy, etc.), it should be possible to build a model to calculate how many states should be used and with what delays. The states and synchronization delays could also be adjusted dynamically by the servers, adapting to changing network conditions. This aspect of TSS has not been fully explored, we presently select parameters which we think may be good choices.

4. PERFORMANCE EVALUATION

4.1 Implementation

As part a project studying the feasibility of mirrored server architectures, TSS was implemented in the open-source QuakeForge [11] server for the Quake I first-person shooter. In these experiments there are 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, approximately simulating TSS operating at the client in a fully distributed version of Quake. The latency of the server-to-server connection is 50 ms.

The first step in implementing TSS in Quake was to alter the server so that all game state data was within a context, and create functions capable of copying one 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 dereferencing of pointers. 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 per-state pseudo-random number generator (RNG) with the same initial seed in each state would be used to provide consistent random numbers in each state. On rollbacks, the RNG state is included in the copied state in case the leading state had incorrectly used more random numbers. Unfortunately, Quake was not designed this way and preserving random events turned out to be a major difficulty in implementing and evaluating TSS.

Once the Quake server had been altered to use contexts, it was fairly simple to add synchronization. Instead of executing client packets immediately, they are intercepted and sent out over a multicast channel. Upon receiving a multicast command, it is inserted back into Quake's network buffer and parsed by the normal Quake functions. In the main event loop, each of the states is checked. Any pending commands that are ready are executed, and inconsistencies between the executing trailing state and its leading state are checked for. In addition to mirroring and synchronization, we also added a trace feature to the 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.

4.2 Simulation

To test the performance of TSS, we ran a series of simulations using the trace feature described above with different network and synchronization parameters. Unfortunately, due to difficulties accounting for all random events, inconsistencies often occurred when they shouldn't have. Even in cases where no inconsistencies should occur, such as with a single server and two states, where there are no remote commands to arrive late, numerous inconsistencies were seen upon simulation. Hence, detailed and accurate study of the behavior of TSS with different configurations and different network conditions on the private multicast network has not been performed because of this.

Nevertheless, we were able to gather some useful results on the cost of rollbacks in Quake. The results in Table 1 show seven runs, all using the same trace file with three users connected to each of two mirrors. The statistics were gathered at mirror one, which saw 18,593 commands from local clients and the multicast group. 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 context copy, repairing data structures within the context, and moving events back to the event queue to be re-executed. "Total Time" is for the entire server, which includes command execution, rollback, and other functions of the server such as providing reliable multicast. Finally, per-command and per-rollback costs are calculated by dividing the execution time and rollback time by number of commands and rollbacks respectively.

The command cost is dominated by the actual execution of the command in the Quake simulator. The 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 context and repair data structures, while the time spent moving executed commands back to the event queue for re-execution is minor. In all seven of the runs these costs were nearly identical, with command execution being an order of magnitude less expensive than rollback. The large jumps in number of commands executed and rollbacks incurred for the fourth and seventh runs are due to a combination of the large gap between the last two states combined with the spurious inconsistencies due to the RNG (there should not have been any real inconsistencies later than 50 ms in these simulations with a 50 ms server-to-server latency). In both of these runs, any inconsistencies between the final two states required 900 ms and 500 ms worth of commands to be re-executed after a rollback. Similarly, each of these re-executed commands was subject to the same RNG caused inconsistencies, so there were additional cascading rollbacks generated in these two runs as well.

Despite the inability to obtain full simulation results, these results are promising for the suitability of TSS in first-person shooters. Performing a snapshots, which involves the same basic tasks as a rollback, is an order of magnitude more expensive than executing a command in Quake. From our analysis in Section 3.2, TSS should be able to perform well in these games if the difference in synchronization delays between states is not too large.

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, this form of synchronization can perform well in high-speed games where there is a large game state and many commands to be synchronized. There are still a number of unexplored areas for future work. As discussed above, a more thorough examination of the parameter space for synchronization delays is needed, as well as dynamically determining the number of, and delays for, needed states. Additionally, building off the distinction between weakly and strongly consistent events, it would be interesting to look at the effect of compensating for weak inconsistencies in future states as opposed to performing a full rollback as the current implementation does.

6. REFERENCES

[1] N. Baughman and B. Levine. Cheat-proof playout for centralized and distributed online games. In *Proc. Infocom 2001*, April 2001.

[2] E. Cronin, B. Filstrup, and A.R. Kurc. A distributed multiplayer game server system. UM EECS589 Course Project Report, http://www.eecs.umich.edu/∼bfilstru/quakefinal.pdf, May 2001.

[3] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A global Internet host distance estimation service. *IEEE/ACM Transactions on Networking*, 9(5):525–540, October 2001.

[4] T.A. Funkhouser. RING: A client-server system for multiuser virtual environments. In *Proc. 1995 Symposium on Interactive 3D Graphics*, pages 85–92. ACM SIGGRAPH, April 1995.

[5] L. Gautier, C. Diot, and J. Kurose. End-to-end transmission control mechanisms for multiparty interactive applications on the Internet. In *Proc. of IEEE Infocom 1999*, volume 3, March 1999.

[6] D. A. Helder and S. Jamin. End-host multicast communication using switch-tree protocols. In *Proc. of GP2PC*, May 2002.

[7] id Software. Quake. `http://www.idsoftware.com/`.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[9] P. Lincroft. The Internet sucks: Or, what I learned coding *X-Wing vs. TIE Fighter*. In *Proc. of Game Developers Conference 1999*, March 1999.

[10] M. Mauve. How to keep a dead man from shooting. In *Proc. of the 7th International Workshop on Interactive Distributed Multimedia Systems*, pages 199–204, October 2000.

[11] The QuakeForge Project. QuakeForge. `http://www.quakeforge.net/`.

[12] J. S. Steinman, R. Bagrodia, and D. Jefferson. Breathing time warp. In *Proc. of the 1993 Workshop on Parallel and Distributed Simulation*, pages 109–118, May 1993.

[13] J. S. Steinman, J. W. Wallace, D. Davani, and D. Elizandro. Scalable distributed military simulations using the SPEEDES object-oriented simulation framework. In *Proc. of Object-Oriented Simulation Conference (OOS'98)*, pages 3–23, 1998.