A Demonstration Of a Lockless, Relaxed Atomicity State Parallel Game Server (LEARS)

Kjetil Raaen, Håvard Espeland, Håkon Kvale Stensland, Andreas Petlund, Pål Halvorsen, Carsten Griwodz
NITH, Norway Simula Research Laboratory, Norway IFI, University of Oslo, Norway
Email: raakje@nith.no, {haavares, haakonks, apetlund, paalh, griff}@ifi.uio.no

Abstract—**Games where thousands of players can interact concurrently poses many challenges with regards to the massive parallelism. Earlier work within the field suggests that this is difficult due to synchronization issues. In this paper, we present an implementation of a game server architecture based on a model that allows for massive parallelism. The system is evaluated using traces from live game sessions that has been scaled up to generate massive workloads. We measure the differences in server response time for all objects that need timely updates. We also measure how the response time for the multithreaded implementation varies with the number of threads used. Our results show that the case of implementing a game-server can actually be an embarrassingly parallel problem.**

I. Introduction

Online multi-player gaming has experienced an amazing growth. Providers of the popular online games must deliver a reliable service to thousands of concurrent players meeting strict processing deadlines in order for the players to have an acceptable quality of experience (QoE). In order to achieve this, game worlds are partitioned into areas-of-interest to minimize message passing between players with no interaction, and allow the game world to be divided between servers. However, players still tend to cluster together at interesting spots making dynamic area rescaling and migration between servers challenging.

There is a prevailing belief in the need for single threaded execution of the game event loop in order to preserve what is seen as critical dependencies in the game [2]. Due to this constriction, even when provisioning for scalability in designing an online game, service providers still find that the processing power on the server side is becoming scarce [4]. Latency is an important metric for online multiplayer games. Claypool at al. [5] classify different types of games, and conclude that for first person shooter (FPS) and racing games, the threshold for an acceptable latency is 100ms. Furthermore, Abdelkhalek et al. [3] discuss the behavior and performance of multiplayer game servers. They find that in terms of benchmarking methodology, game servers are very different from other scientific workloads. Most of the sequentially implemented game servers can only support a limited numbers of players, and the bottlenecks in the servers are both game-related and network-related. In [2], the authors extend their work and use the computer game Quake to study the behavior of parallelism. When running on a server with up to eight processing cores the game suffers because of lock synchronization during request processing. High wait times due to workload imbalances at global synchronization points are also a challenge.

The industry is now experimenting with implementations that allow for a greater level of parallelization as a response to the lack of growth of single-processor hardware speedup. We take this model even further and propose a design that allows a game server to be classified as an embarrassingly parallel workload.

In this demonstration, we show a game server model that allows for better resource utilization of multi-processor systems. We have implemented a prototype game using this design. The multithreaded implementation is compared with a single threaded implementation in order to measure the overhead of parallelizing the implementation and showing the experienced benefits of parallelization. Our results indicate that it is possible to design a game server to make it embarrassingly parallel. We can also see that the implementation is able to handle the quadratic increase of in- server communication that develops as many players interact in a game world hotspot.

II. Design and Implementation

Traditionally, game servers have been implemented much like game clients. They are based around a main loop, which updates every active element in the game. These elements include for example player characters, non-player characters and projectiles. The simulated world has a list of all the active elements in the game, and typically calls an "update" method on each element. In this method, the active element will perform all its actions for the time-slot. Since these are sequential operations all actions can be performed directly. The character reads input from the network, performs updates on itself according to the input, and updates other elements with the results of its actions.

To make a parallel game server with minimal locking, the system needs to be redesigned from the ground up with parallelism in mind. The foundations for our parallel approach are thread pools and blocking queues.

A. Parallel main loop

The system demonstrated here uses a threadpool executor as the core of the main loop. When an active element is created in the world, it is scheduled for execution by the thread pool executor. When it executes, the active element updates its state exactly as in the single threaded case. When the task is finished for one time slot, it can reschedule itself for the next slot,

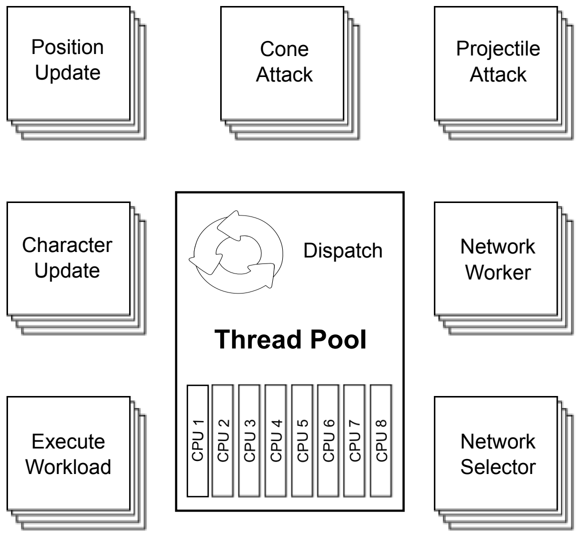


Figure 1. Design of the Game Server

delayed by a specified time. This allows active elements to have any lifetime from one-shot executions to the duration of the program. It also allows different elements to be updated at different rates, depending on the requirements of the game developer.

The demo setup allows the user to interactively change the number of threads in the threadpool. By setting the number of threads below the number of cores it is, by means of the real-time monitoring, possible to examine how well the cores are utilized.

B. Use of locking

The thread pool executor used as described above does not constrain which tasks are executed in parallel. Therefore, all systems elements must allow any of the other elements to execute concurrently. This places one simple constraint on what an element can do; i.e., *Elements can only update their own state, but read any state.*

This is sufficient because there is no need to establish consistent ordering of events. This might sound counter intuitive, but it works since the gameworld already is a simple simulation with a whole series of approximations. All clients are connected to the server through individual network links, with vastly different latencies. The clients are also run on different computers, with different update frequencies. If player A performs an action slightly before player B, this ordering is not necessarily kept by the server in the multi-threaded case. Running client update requests in parallel does not aggravate this problem.

Another potential reason for locking is to keep state changes atomic. This is also unnecessary due to the nature of the problem. Consider the following example: Character A moves while character B attacks. If only the X coordinate of character A is updated at the point in time when the attack is executed, the attack will see character A at position $(X_{t+\Delta T}, Y_t)$. This position is within the accuracy of the simulation which in any case is no better than the distance an object can move within the timeline ΔT. The only requirement for this to work is that assignment operations are atomic.

The end result of our proposed design philosophy is that there is no synchronization in the server under normal running conditions.

C. Message passing

Blocking queues [1] are queue implementations that are synchronized separately at each end. This means that elements can be removed from the queue simultaneously with elements being added. Each of these operations is also extremely quick, so the probability of blocking is low.

These are used by the implementation to allow information to be passed between active objects. Each active object that can be influenced by others has a blocking queue of messages. During its update, it will read and process the pending messages from its queue. Other active elements put messages in the queue to be processed when they need to change the state of other elements in the game.

III. Example Game

This demonstration uses an implementation of a very simple game which nonetheless contains all typical elements of a full Massively Multiplayer Online game, with the exception of persistent state. The game itself is simple. Each player controls a small circle ("the character") with an indicator for which direction they are heading. The characters are moved around by pressing keyboard buttons. They also have two attacks: One projectile and one instant area of effect attack. Both attacks are aimed straight ahead. If an attack hits another player character, the attacker gets a positive point, and the character that was hit gets a negative point. This simple game provides examples of all the elements of the design described above:

- The player character is a long lifetime active object. It processes messages from clients, updates states and potentially produces other active objects (attacks). In addition to position, which all objects have, the player also has information about how many times it has been hit and how many times it has hit others as part of its state. The player character also has a message queue to receive messages from other active objects. At the end of its update, it will enqueue itself for the next update unless the client has disconnected.
- The frontal attack is a one shot task that finds player characters in its designated area and sends messages to those hit so they can update their counters, as well as back to the attacking player informing about how many were hit.
- The projectile is a short lifetime object that moves in the world, checks if it has hit anything and reschedules itself for another update, unless it has hit something or ran to the end of its range. The projectile can only hit one target.

Our implementation has no spatial partitioning, we are interested in the worst case, hence the numbers presented in this paper assume all players can see each other at all times. This means that the number of messages and interactions by necessity grows by the number of clients squared.

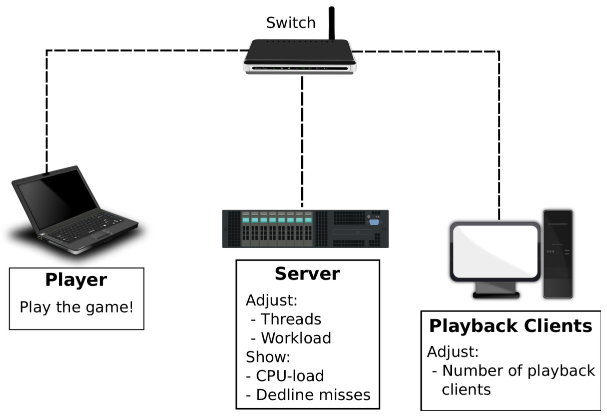


Figure 2. Setup of the demonstration

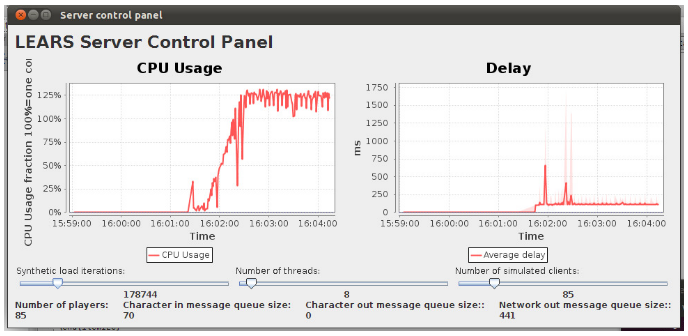
To simulate workload that grow linearly with number of players, especially collision checks with the ground and other static objects, we have included a synthetic load. The synthetic load is designed to emulate collision checks with a grid representing the ground in the virtual world. To achieve this, we have created an array of floating point values representing a part of the gameworld. For each scheduled update, each character has to perform a square operation on a given number of elements in the array. The operation is seeded with a randomly generated value in order to avoid runtime optimizations in the virtual machine. Which elements are processed depends on the player's position in the gameworld. How many array elements are processed determines the severity of the load. By adding the synthetic load, the cache is dirtied and the game server processes becomes more realistic compared to a large-scale MMORPG. The demo setup allows the user to experiment with different levels of synthetic load.

IV. Demonstration

The demo setup consists of three machines and is illustrated in figure 2. One machine is the game server. For this purpose, we are using a four-core CPU, which gives us enough parallel processing power to illustrate improvements over single threaded implementations. Another machine plays back a trace of actual gameplay in as many instances needed to emulate high loads on the server. The client playback computer has no user interface, as the number of simulated clients is controlled from the server control panel. The client simulation machine is not a bottleneck, since simulating clients requires very little calculation. The last machine is for running the real game client, so the experimenter can see what is going on on the server and play the actual game.

The demo setup also includes a wireless router and and a web-server for sharing the game client with interested attendees. This setup allows others to download the client and play the game during the demo session.

For the purposes of this demonstration, the game server has been updated with an interactive graphical user interface. Using this interface, an experimenter can configure the conditions for the experiment on the fly, while watching the results in real-time.



Synthetic load iterations: 178744
Number of threads: 8
Number of simulated clients: 85
Number of players: 85
Character in message queue size: 70
Character out message queue size:: 0
Network out message queue size:: 441

Figure 3. Screenshot of the game server GUI

The GUI, shown in figure 3 has two plots, both updated once every second. The left plot shows CPU usage vs. time. This allows users to monitor the achieved CPU utilization under the current running conditions. The right plot displays the interval between updates. The system is designed to do an update every 100ms. Increases above this level is considered deadline misses. The red line shows the average value for the current measurement interval, the pink outline shows the highest and lowest values. This is the main performance metric for the server system, and any value above approximately 250ms will severely deteriorate the QoE for players of the game.

Below the plots there are two sliders. One controls the number of iterations for the synthetic load. The experimenter can adjust this to change the intesnity of the load generated by each player. The next slider sets the number of threads available to the thread pool. The final slider instructs the playback client to start or stop playback instances in order to reach the chosen number. This slider has a delay before activation, and will be disabled until the requested number of clients have had time to connect to the game.

Using this setup it is possible to study, in real-time, how our Lockless, Relaxed Atomicity State Parallel Game Server (LEARS) handles different conditions.

Acknowledgements

This work has been performed in the context of the *iAD* centre for Research-based Innovation (project number 174867) funded by the Norwegian Research Council.

References

[1] Java Platform, Standard Edition 6 API Specification, class BlockingQueue. http://download.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html.

[2] A. Abdelkhalek and A. Bilas. Parallelization and performance of interactive multiplayer game servers. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, page 72, Washington, DC, USA, april 2004. IEEE Computer Society.

[3] A. Abdelkhalek, A. Bilas, and A. Moshovos. Behavior and performance of interactive multi-player game servers. *Cluster Computing*, 6:355–366, October 2003.

[4] W. Cai, P. Xavier, S. J. Turner, and B.-S. Lee. A scalable architecture for supporting interactive games on the internet. In *Proceedings of the sixteenth workshop on Parallel and distributed simulation*, PADS '02, pages 60–67, Washington, DC, USA, 2002. IEEE Computer Society.

[5] M. Claypool and K. Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, Nov. 2005.