LEARS: A Lockless, Relaxed Atomicity State Model for Parallel Execution in Game Servers

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*—Virtual worlds where thousands of players can interact concurrently poses many challenges. For a modern, multicore server to host as many players as possible for such a virtual world, the server software has to implement support for 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, when the system is built from the ground up with parallelism in mind, the case of a game-server can actually be an embarrassingly parallel problem.**

I. Introduction

Over the last decade, 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).

One major goal for such game providers is to support as many concurrent players in a gameworld as possible. In order to achieve this, gameworlds are partitioned into areas-of-interest to minimize message passing between players with no interaction, and allow for the gameworld to be divided between servers. This approach is however limited by the distribution of players in the gameworld. According to [**?**], distribution of players is heavy-tailed, with about 30% of players in 1% of the game area. This paper focuses on how many players can be handled in a single segment of the game world.

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 [**?**]. 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 [**?**].

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

In this paper we present a model of design that allows for better resource utilization of multi-processor systems at the game server. We have implemented a prototype game using this design, and evaluated it using captured traces from real gameplay in order to generate load for the server. 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. We also study how the responsiveness of the different implementations change with increased load on the server. We discuss generic elements of this game design, and impacts of our chosen platform of implementation.

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 gameworld hotspot.

II. Basic Idea

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 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. The simulated time is kept constant throughout each iteration of the loop, so that all elements will get updates at the same points in simulated time. Since only one element updates at once, 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 main concept behind the LEARS approach is to split the game server executable into threads on the finest possible granularity. Each update of every player character, AI opponent and projectile runs as an independent work unit. Using this approach the theoretical parallelism will be proportional to the load on the server. The fine granularity creates a need for significant communication between threads. To avoid locking as much as possible, the system follows one core rule:

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 in-

tuitive, but 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 single 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 ΔT. The only requirement for this to work is that assignment operations are atomic.

One exception from this rule is actual trades among players. This situation is outside the scope of this paper for two reasons: 1) the atomicity of trades should, by best practice, be enforced by the underlying database implementation and not the game server, 2) trades are rare situations and usually occur far away from the action, and will therefore impact performance minimally.

The design described in this paper does not use spatial partitioning. As described in section V-A 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.

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

III. Design and Implementation

The foundations for the parallel approach are thread pools, and blocking queues.

A. Thread pool

Creation and deletion of threads incurs overhead in current systems, so threads should be kept as long as possible. Context switching is also an expensive operation, so the number of threads should not grow unbounded. These issues constrain how a system can be designed. The thread pool pattern is designed to work around these constraints. A thread pool executor [**?**] is a system that maintains a pool of threads, and a queue of tasks. When a thread is available, the executor will pick a task from the queue and execute it. The thread pool system itself is not preemptive, so the thread will run each task until it is done. This means that in contrast to normal threading, each task should be as small as possible. Larger units of work should be split up into several sub-tasks.

The thread pool implementation used in this paper supports the concept of delayed execution. This means that tasks can be put into the work queue for execution at a time specified in the future.

The thread pool presents itself as a good way to balance the number of used threads when the work is split into extremely small units. When an active element is created in the world, it will be scheduled for execution by the thread pool executor. When it executes, the active element will update 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, 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.

B. Use of locking

The thread pool executor used as described above will not constrain which tasks are executed in parallel, therefore all systems elements must allow any of the other elements to execute concurrently.

C. Blocking queues

Blocking queues [**?**] are queue implementations that are synchronized separately at each end. This means that elements can be removed form 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.

D. Our implementation

As there are, to our knowledge, no implementations of this design openly available, our prototype implementation was made for this paper. Our implementation is a very simple game which nonetheless contains all elements of a full Massively Multiplayer Online game, with the exception of persistent state.

Persistent state would introduce some complications, but these are left to future work, as database transactions are not time critical can usually be scheduled outside peak load situations.

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:

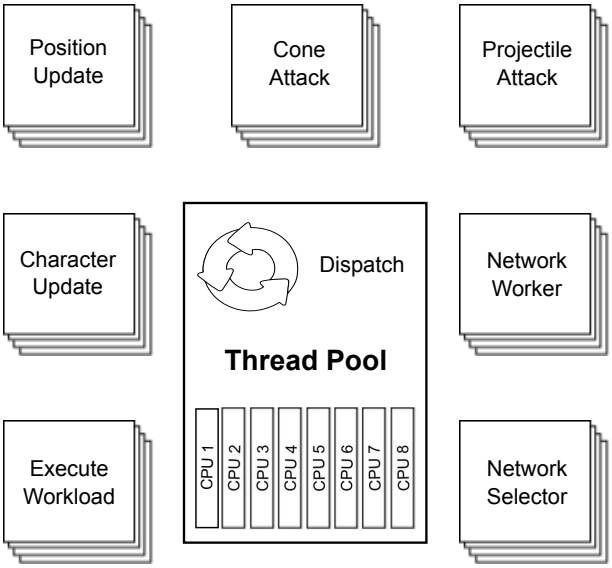


Figure 1. Design of the Game Server

- 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 another the next update unless the client it represents has disconnected.
- The frontal cone 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.

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.

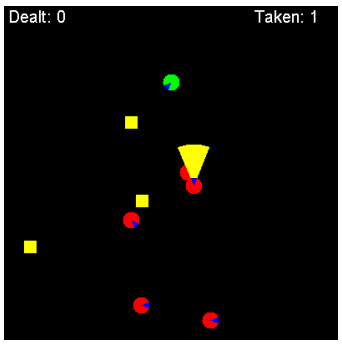


Figure 2. Screen shot of game with six players.

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. For our experiments, 1000 array elements were processed for each character update, emulating collision detection with a high-resolution terrain mesh.

The system described in this paper is implemented in Java. This programming language has strong support for multithreading and has well-tested implementations of all the required components. The absolute values resulting from these experiments depend strongly on the complexity of the game, as a more complex game would require more processing. In addition the absolute values will depend on the runtime environment, especially the server hardware. The choice of programming language can also influence absolute results from the experiments. However, the focus of this paper is the relative results, as we are interested in comparing scalability of the multithreaded solution with a single-threaded approach and whether the multithreaded implementation can handle the quadratic increase in traffic as new players join.

IV. Evaluation

A. Experiment setup

To load the server in a realistic way, the game was run once with 5 human players playing the game. The network input to the server from this session was recorded with a timestamp for each message. This information could then be played back to simulate clients. Playing it back repeatedly in parallel with a slight offset to the start time allows us to simulate an arbitrary number of player on the server. To ensure that client performance is not a bottleneck, the simulated clients were distributed among multiple physical machines.

The local 1Gbps network these clients are transmitting across is not a limiting factor on performance. One client will on average generate 2.6kbps of traffic. The game server was run on a server machine containing 4 Dual-Core AMD Opteron 8218, at 2600MHz and 16GB of RAM. To ensure comparable numbers, the server was taken down between each test run.

B. Response latency

The most important metric for the performance of a realtime game server is the time between updates for each player. This time is an important part of the response time as seen from the player. (The other main contributor is network latency.) The time between updates also determines the spatial resolution of the game simulation, which if it becomes too low, will be noticeable by players. Significant work is done in the game industry to hide server response latency from the players, but at some point it will always degrade user experience above a certain level.

For this work, we set the time between updates to 100 ms. Thus, an optimally running server will update each game

element once each 100 ms. This update interval should allow for a good experience for players in most classes of networked games [**?**]. The server is configured to not run any faster than this, as that would create unnecessary network traffic. Update times longer than this indicate that the server is starved for resources. Experience indicates that rare delay peaks of less than a second will still be acceptable (as such short events are easily hidden at the client), but if the numbers rise persistently, the server should be considered overloaded.

The experiment described in section IV-A was run with client numbers from 40 to 800 with increments of 40. Figure 3 shows the response time statistics from these experiments. Each pair of boxes has data from a single-threaded run on the left, and a multithreaded run to the right. They all use a pool of 48 worker threads, and distributes connections across 8 ports. Boxes from left to right describe increasing number of clients.

From this plot, we can see that the single-threaded implementation is struggling to keep up at 280 players. The median is 299ms and it already has outliers all the way to 860ms.

The multithreaded server is handling the players well up to 640 players where we are getting outliers above 1 second, and the median is at 149ms.

These statistics might get influenced by the fact that the number of samples is proportional to the update frequency. This means that long update cycles to a certain degree get artificially lower weight. This effect should be minimal though, as we have a large amount of samples in from each run.

Figure IV-B shows details of two interesting cases. In figure 4(a), the case for 400 players is shown, where the single-threaded server is missing its deadlines, while the multithreaded version is processing almost everything on time. At 800 players as shown in figure 4(b), the outliers are going much further for both cases. Here, even the multithreaded implementation is struggling to keep up, though it is still handling the load significantly better than the singlethreaded version, which is generally compeletely unplayable.

C. Resource consumption

Analyzing the the CPU load data for 620 players, we see that the CPU us slightly overloaded. The mean response time is 133ms, above the ideal delay of 100ms. Still, the server is able to keep the update rate smooth, without significant spikes. The CPU utilization grows while the clients are logging on, then stabelizes at an allmost full CPU utilization for the rest of the run. The two spikes in response time happen while new players log in to the server at a very fast rate (30 clients pr. second). Recieving a new player requires a lock in the server, hence this operations is to a certain degree serial.

D. Effects of thread-pool size

In 6 we see clearly that the system utilizes more than 4 cores efficiently, as the 4 thread version shows significantly higher response times. At one thread per core or more, the numbers are relatively stable, with a tendency towards more consistent low response times with more available threads. This could

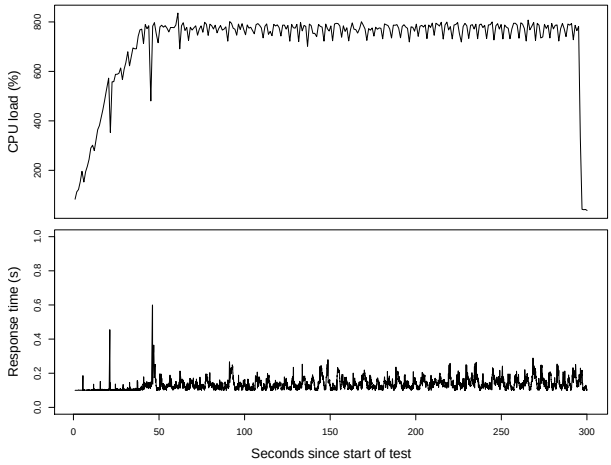


Figure 5. CPU load for 800 concurrent clients on the multithreaded server.

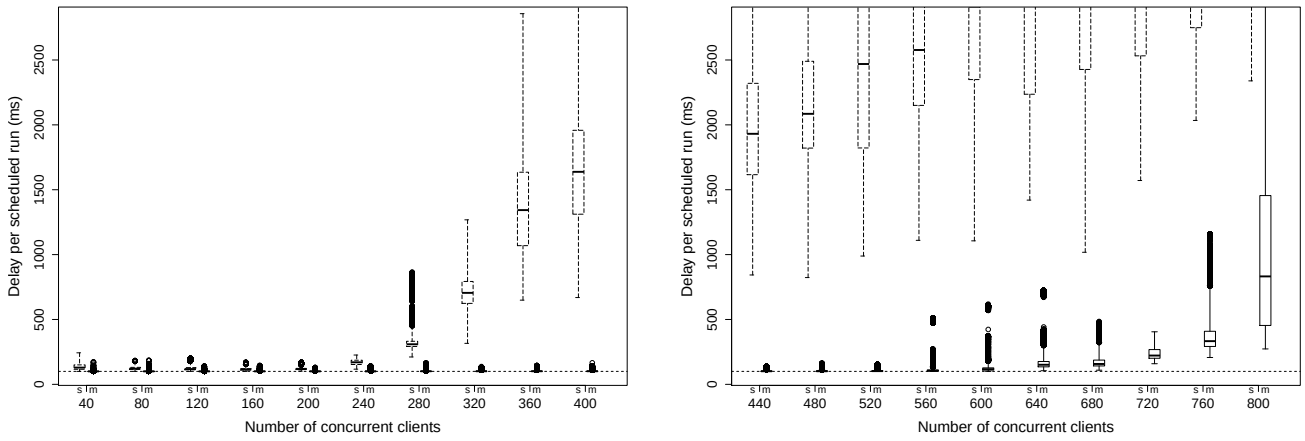
Figure 6. Response time for 800 concurrent clients on using varying number of threads.

mean that threads are occasionally waiting for I/O operations. Since thread pools are not preemptive such situations would lead to one core going idle if there are no other available threads. Too many threads, on the other hand, could lead to excessive context switch overhead.

V. Discussion

A. Spatial partitioning

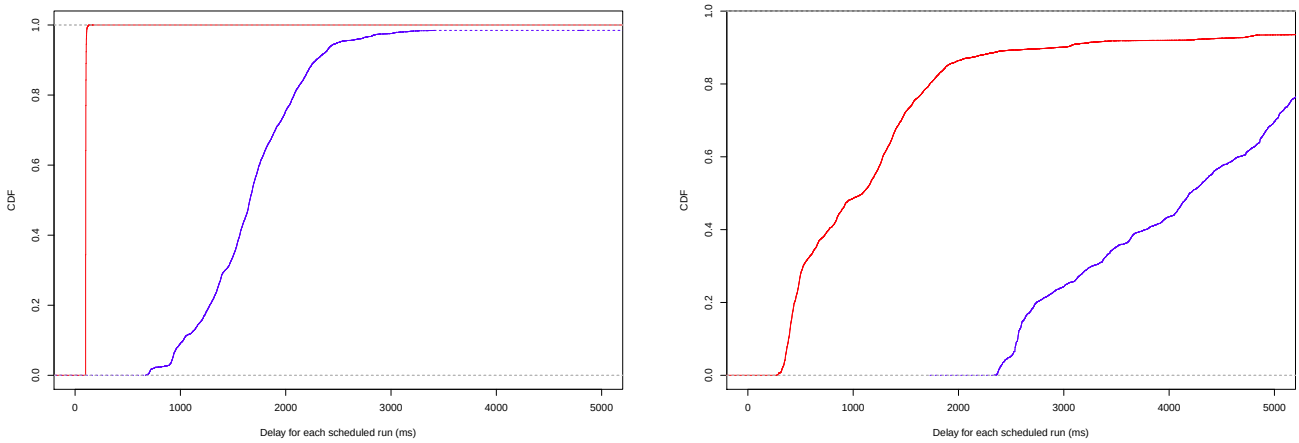
Most approaches in the literature to multithreaded game server implementations including the implementation in [**?**] use some form of spatial partitioning to lock parts of the game world while allowing separate parts to run in parallel. Spatial partitioning is also used in other situations to limit workload. The number of players game designers can allow in one area in a game server is limited by the worst-case scenario. The worst case scenario for a spatially partitioned game world is when everybody move to the same point, where the spatial partitioning still ends up with everybody in the same partition regardless of granularity. This worst case is not as uncommon as it may seem, as people tend to congregate if the get the chance. Hence, this paper will not consider spatial partitioning, as we want to investigate the worst case scenario.



(a) 40-400 concurrent clients

(b) 440-800 concurrent clients

Figure 3. Response time for single threaded and multithreaded server with increasing number of concurrent clients. (Note the difference in time scales.)



(a) 400 concurrent clients

(b) 800 concurrent clients

Figure 4. CDF of response time for single threaded and multithreaded server with 400 and 800 concurrent clients respectively.

VI. Related Work

There are some earlier research on how to optimize game server architectures for online games, both MMOs and smaller-scale games. In this section we summarize some of the most important findings from related research in this field.

Cai et al. [**?**] presents a scalable architecture for supporting large-scale interactive Internet games. Their approach divide the game world into multiple partitions and assign each partition to a server. The solution is more scalable than a single centralized server, and compared to a fully distributed peer-to-peer architecture, the game developers can detect malicious behavior. The issues with this solution is that the architecture of the game server is still a limiting factor. Only a limited number of players can interact in the same server partition at a given time. There have been proposed several middleware systems for automatically distributing the game state among several participants. In [**?**] the authors present a middleware called RTF which allows game developers to create large, seamless virtual worlds and to migrate zones between servers. Beskow et al. [**?**] have also been investigating partitioning and migration of game servers. Their approach uses core selection algorithms to locate the most optimal server. This approach does, however, not solve the challenge of many players that want to interact in a popular area. In [**?**] the authors discuss the possibility of using grids as servers for massively multiplayer online games (MMOG). Their research shows that proxy servers are needed to scale the number of players in the game.

Latency is an important metric for online multiplayer games. In [**?**] the authors are classifying different types of games, and conclude that for first person shooter (FPS) and racing games, the thresholds for an acceptable latency is 100ms. For other classes of networked games, like real-time strategy (RTS) and Massively multiplayer online games (MMOGs) players will tolerate somewhat higher delays, but there are still strict latency requirements in order to provide a good QoE. The accumulated latency of network transmission, server processing and client processing adds up to the latencies that

the user is experiencing, and reducing any of these latencies will improve the user's experience.

In [**?**] the authors are discussing the behavior and performance of multiplayer game servers. They find that in the 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 [**?**] the author extend their work and uses the computer game Quake to study the behavior. 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.

There have been a lot of research on how to optimize the network model with respect to latency, and a lot of research has also been done on how to partition the server and scale the number of players by offloading to several servers. Modern game servers have also been parallelized to scale with more processors, however, a large amount of processing time is still wasted on lock synchronization. In our game server design, we try to eliminate the global synchronization points and locks, thus making the game server an embarrassingly parallel workload. In the next section we will describe the design of our game server.

VII. Conclusion

A. Conclusion

In this paper we have shown that, if designed from the ground up with parallelism in mind, game servers can scale well with the number of cores on a unified memory multiprocessor system, even in the case where all players must be aware of all other players and their actions. The threadpool system balances load well between the cores, and its queue-based nature means that no task is starved unless the entire system lacks resources. Message passing through the blocking queue allows objects to communicate intensively without blocking each other. We show that the 8-core server can handle a factor of 4 more clients utilizing the multithreaded model before the response time becomes unacceptable. Considering that a significant amount of the load scales quadratically with the number of players, this is close to the theoretical best case, placing the problem of game servers in the class of embarrassingly parallel problems.

We found that the limiting factor in the current implementation is the current garbage collector. These results all depend on the concept that state changes in a game need not be atomic.

B. Future work

From the research described in this paper, a series of further experiments present themselves.

The relationship between linearly scaling load and quadratic load can be tweaked in our implementation. This could answer questions about which type of load scale better under multithreaded implementations.

Another direction this work could be extended is to go beyond the single shared memory computer used here, and distribute the workload across clusters of computers. This could be achieved by implementing cross-server communication directly in the server code, or by utilizing existing technology that makes cluster behave like shared memory machines.

All experiments described here were run with an update frequency of 10Hz. This is good for many types of games, but different frequencies are relevant for different games. Investigating the effects of running with a higher or lower frequency of updates on server performance could yield interesting results.

If, during the implementation of a complex game, it is shown that some state changes must be atomic to keep the game state consistent, the message passing nature of this implementation means that we can use read-write-locks for any required blocking. If such cases are found investigating how read-write-locking influence performance would be worthwhile.