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

Kjetil Raaen, Håvard Espeland, Håkon K. 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

Supporting thousands of interacting players in a virtual world poses huge challenges with respect to processing. To host as many players as possible, the server software has to implement support for massive parallelism. Earlier work within the field suggests that this is difficult due to synchronization issues, but in this paper, we present the design and implementation of a game server architecture based on a model that allows for massive parallelism. Our prototype is evaluated using traces from live game sessions where we measure the server response time for all objects that need timely updates. We also measure how the response time for the multi-threaded implementation varies with the number of threads used. Our results show that the challenge of scaling up a game-server can be an embarrassingly parallel problem.

1. INTRODUCTION

One major goal for large game providers is to support as many concurrent players in a game-world as possible while preserving the strict latency requirements in order for the players to have an acceptable quality of experience (QoE). In order to achieve this, game-worlds are typically partitioned into areas-of-interest to minimize message passing between players with no interaction and to allow the game-world to be divided between servers. This approach is however limited by the distribution of players in the game-world, and the problem is that the distribution of players is heavy-tailed with about 30% of players in 1% of the game area [4].

In such scenarios, the important metric for online multi-player games is latency. Claypool et. al. [6] classify different types of games and conclude that for first person shooter (FPS) and racing games, the threshold for an acceptable latency is 100ms. For other classes of networked games, like real-time strategy (RTS) and massively multi-player 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.

The traditional design of massively multi-player game servers rely *sharding* for scalability beyond what a single CPU core can handle. *Sharding* involves making a new copy of an area of a game, where players in different copies are unable to interact. This approach eliminates most requirements for communication between the processes running individual shards. An example of such a design can be found in [5].

The industry is now experimenting with implementations that allow for a greater level of parallelization. One known example is Eve Online [7]. With LEARS, we take this approach even further and focus on how many players can be handled in a single segment of the game world. We present a model that allows for better resource utilization of multi-processor, game server systems which should not replace spatial partitioning techniques for work distribution, but rather complement them to improve on their limitations. Furthermore, a real prototype game is used for evaluation where captured traces are used to generate server load. We compare multi-threaded and single-threaded implementations in order to measure the overhead of parallelizing the implementation and showing the experienced benefits of parallelization. The change in responsiveness of different implementations with increased load on the server is studied, and we discuss generic elements of this game design which impact of our chosen platform of implementation.

Our results indicate that it is possible to design an "embarrassingly parallel" game server. We also observe that the implementation is able to handle a quadratic increase of in-server communication when many players interact in a game-world hotspot.

2. LEARS: THE 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 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. 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. This point in time is refered to as a *tick*. Using this method, the active element will perform all its actions for the tick. Since only one element updates at a time, 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.

LEARS is a game server model with support for lockless, relaxed atomicity state parallel execution. The main concept is to split the game server executable into lightweight threads at 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. To do this, we must relax the presumed deterministic requirements of a game server, and we will show that this approach retains consistency and is applicable to real-world games.

Our claim is that the ordering of events scheduled to execute at a tick does not always need to be considered. This is the case for many games and is mainly an issue of game design, i.e., what is the desired behavior if two players perform conflicting actions at the same instant. In the traditional main-loop approach, every event in a game scheduled for a tick are executed by the main loop. The main loop process these in arrival order. Thus, the ordering is highly influenced by the client latencies and at which point in time between two ticks the event was dispatched by the client. Remember that a tick is the smallest amount of time considered by the game, and this means there is no correct order to execute conflicting events within a single tick. As such, the ordering of events scheduled for a tick in a traditional main loop is *not* deterministic. LEARS takes advantage of this relaxation and allows events scheduled for a tick to execute in any order.

The second relaxation relates to game state consistency. The fine granularity creates a need for significant communication between threads to avoid problematic lock contentions. Systems where elements can only update their own state and read any state without locking [1] will obviously not work in all cases. However, game servers are not accurate simulators, and again, depending on the game design, some (internal) errors are acceptable without violating game state consistency. On the other hand, for actions where a margin of error is not acceptable, transactions can be used keeping the object's state internally consistent. However, locking the state is expensive. Fortunately, most common game actions do not require transactions, an observation that we take advantage of in LEARS.

These two relaxations allow actions to be performed on game objects in any order without global locking. It can be implemented using message passing between threads and retains consistency for most game actions. This includes actions such as moving, shooting, spells and so forth. The end result of our proposed design philosophy is that there is no synchronization in the server under normal running conditions. Since there are cases where transactions are required, they can be implemented outside the LEARS event handler running as transactions requiring locking. In the rest of the paper, we will consider a practical implementation of LEARS, and evaluate its performance and scalability.

3. DESIGN AND IMPLEMENTATION

In our experimental prototype implementation of the LEARS concept, the parallel approach is realized using thread pools and blocking queues.

3.1 Thread pool

Creation and deletion of threads incur large overheads, and context switching is an expensive operation. These overheads constrain how a system can be designed, i.e., threads should be kept as long as possible, and the number of threads should not grow unbounded. We use a *thread pool* pattern to work around these constraints, and a thread pool executor (the Java `ThreadPoolExecutor` class) to maintain the 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, i.e., larger units of work should be split up into several sub-tasks.

The thread pool is a good way to balance the number of threads when the work is split into extremely small units. When an active element is created in the virtual world, it will be scheduled for execution by the thread pool executor, and the active element will update its state exactly as in the single threaded case. Furthermore, our thread pool 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. 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.

3.2 Blocking queues

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

To enable a fast communication between threads with shared memory (and caches), we use *blocking queues*, using the Java `BlockingQueue` class, which implements queues that are synchronized separately at each end. This means that elements can be removed from and added to the queue simultaneously, and since each of these operations are extremely fast, the probability of blocking is low. Thus, these queues 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.

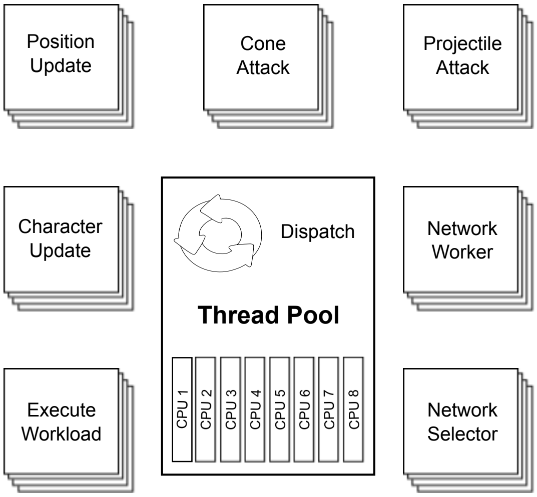


Figure 1: Design of the Game Server

3.3 Our implementation

To demonstrate LEARS, we have implemented a prototype game containing all the basic elements of a full MMOG with the exception of persistent state. Persistent state do introduce some complications, but as database transactions are often not time critical and can usually be scheduled outside peak load situations, we leave this to future work.

In the game, each player controls a small circle ("the character") with an indicator for which direction they are heading (see figure 2). The characters are moved around by pressing keyboard buttons. They also have two types of attack, i.e., 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. The game provides examples of all the elements of the design described above:

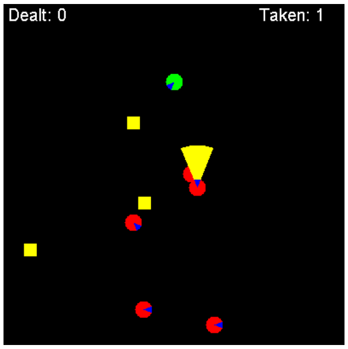


Figure 2: Screen shot of a game with six players.

- 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. 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 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 an MMORPG workload that grow linearly with number of players, especially collision checks with the ground and other static objects, we have included a synthetic load which emulates collision detection with a high-resolution terrain mesh. The synthetic load ensures that the cache is regularly flushed to enhance the realism of our game server prototype compared to a large-scale game server.

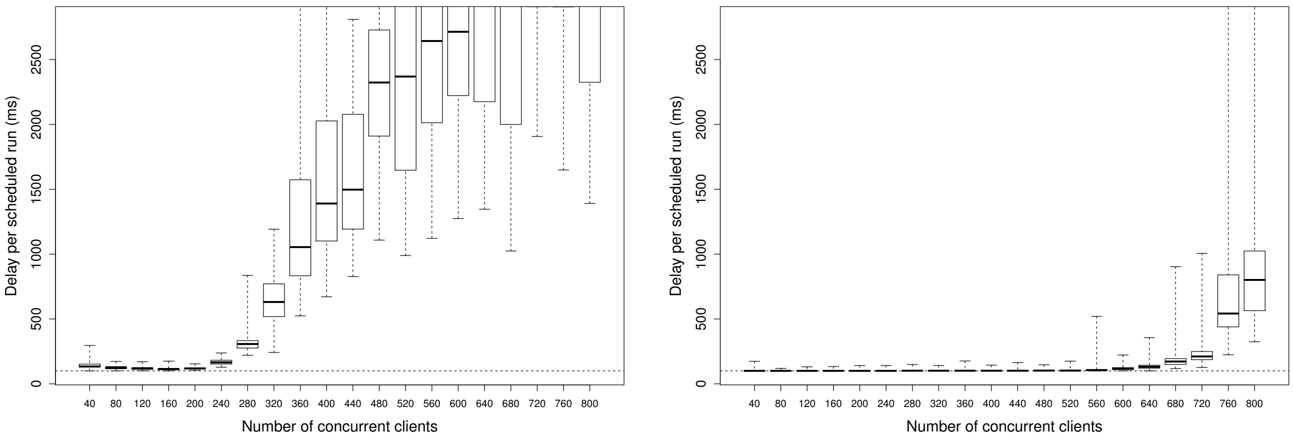
The system described in this paper is implemented in Java. This programming language has strong support for multi-threading and has well-tested implementations of all the required components. The absolute values resulting from these experiments depend strongly on the complexity of the game, as a more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

4. EVALUATION

To have a realistic behavior of the game clients, the game was run with 5 human players playing the game with a game update frequency of 10 Hz. The network input to the server from this session was recorded with a timestamp for each message. The recorded game interactions were then played back multiple times in parallel to simulate a large number of clients. To ensure that client performance is not a bottleneck, the simulated clients were distributed among multiple physical machines. Furthermore, as an average client generates 2.6 kbps network traffic, the 1 Gbps local network interface that was used for the experiments did not limit the performance. The game server was run on a server machine containing 4 Dual-Core AMD Opteron 8218 (2600 MHz) with 16 GB RAM. To ensure comparable numbers, the server was taken down between each test run.

4.1 Response latency

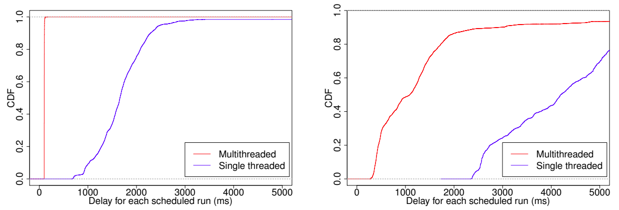
The experiments were run with client numbers ranging from 40 to 800 in increments of 40, where the goal is to keep the latencies below the 100 ms QoE threshold for FPS games [6]. Figure 3 shows a box-plot of the response time statistics from these experiments. All experiments used a



(a) Single-threaded server

(b) Multi-threaded server

Figure 3: Response time for single- and multi-threaded servers (dotted line is the 100 ms threshold).



(a) 400 concurrent clients

(b) 800 concurrent clients

Figure 4: CDF of response time for single- and multi-threaded servers with 400 and 800 concurrent clients.

pool of 48 worker threads and distributed the network connections across 8 IP ports.

From these plots, we can see that the single-threaded implementation is struggling to support 280 players at an average latency below 100 ms. The median response time is 299 ms, and it already has extreme values all the way to 860 ms, exceeding the threshold for a good QoE. The multi-threaded server, on the other hand, is handling the players well up to 640 players where we are getting samples above 1 second, and the median is at 149 ms.

These statistics are somewhat 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.

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

4.2 Resource consumption

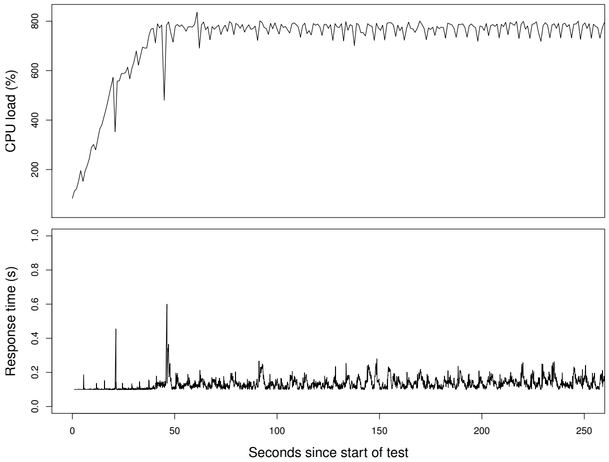


Figure 5: CPU load and response time for 620 concurrent clients on the multi-threaded server.

We have investigated the resource consumption when players connect to the multhreaded server as shown in figure 5. We present the results for 620 players, as this is the highest number of simultaneous players that server handles before significant degradation in performance, as shown in figure 3(b). The mean response time is 133 ms, above the ideal delay of 100 ms. 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 stabilizes at an almost 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). Receiving a new player requires a lock in the server, hence this operation is, to a certain degree, serial.

4.3 Effects of thread-pool size

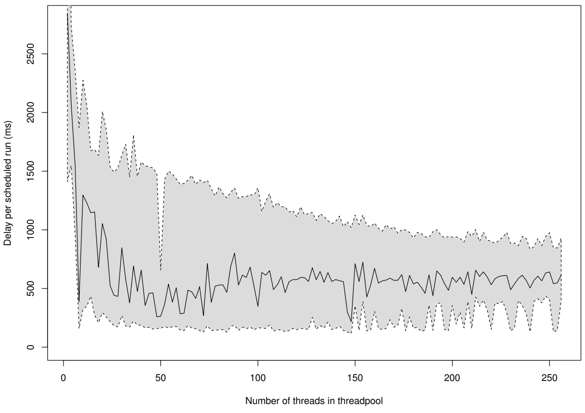


Figure 6: Response time for 700 concurrent clients on using varying number of threads. Shaded area from 5 to 95 percentiles.

To investigate the effects of the number of threads in the threadpool, we performed an experiment where we kept the number of clients constant while varying the number of threads in the pool. 700 clients were chosen, as this number sligtly overloads the server. In figure 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, to about 40 threads. This could 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. The results show that the average is slowly increasing after about 50 threads, though the 95-percentile is still decreasing with increased number of threads.

5. DISCUSSION

Most approaches to multi-threaded game server implementations in the literature (e.g., [1]) 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 that 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 paper investigate an orthogonal and complementary approach which tries to increase the maximum number of users in the worst case scenario where all players can see each other at all times. Thus, spatial partitioning could be added to further scale the game server.

The LEARS approach does have *limitations* and is for example not suitable if the outcome of a message put restrictions on an object's state. This is mainly a game design issue, but situations such as trades can be accommodated by doing full transactions. This is only a problem for trades *within* a single game tick where the result of a message to another object puts a constraint on the original sender, and can be solved by means such as putting the money in escrow until the trade has been resolved, or by doing a transaction outside of LEARS (such as in a database). Moreover, the design also adds some overhead in that the code is somewhat more complex, i.e., all communication between elements in the system needs to go through message queues. The same issue will also create some runtime overhead, but our results still demonstrate a significant benefit in terms of the supported number of clients.

6. RELATED WORK

At Netgames 2011 [10], we presented a demo with a preliminary version of LEARS. There is also some earlier research on how to optimize game server architectures for online games, both MMOGs and smaller-scale games. In this section, we summarize some of the most important findings from related research in this field. For example, "Red Dwarf", the community-based successor to "Project Darkstar" by Sun Microsystems [11], is a good example of a parallel approach to game server design. Here, response time is considered one of the most important metrics for game server performance, and suggests a parallel approach for scaling. The described system uses transactions for all updates to world state, including player position. This differs from LEARS, which invastigates the case for common actions where atomicity of transactions is not necessary.

Work has also been done on scaling games by looking at the optimization as a data management problem. The authors in [12] have developed a highly expressive scripting language called SGL that provide game developers a data-driven AI scheme for non-player characters. By using query processing and indexing techniques, they can efficiently scale to a large number of non-player objects in games. Moreover, Cai et al. [3] present 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 issues with this solution is that the architecture of the game server is still a limiting factor in worst case scenarios as only a limited number of players can interact in the same server partition at a given time. There have also been proposed several middleware systems for automatically distributing the game state among several participants. In [8], the authors present a middleware which allows game developers to create large, seamless virtual worlds and to migrate zones between servers. This approach does, however, not solve the challenge of many players that want to interact in a popular area. The research presented in [9] shows that proxy servers are needed to scale the number of players in the game, while the authors discuss the possibility of using grids as servers for MMOGs.

In [2], the authors are discussing the behavior and performance of multi-player 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. The authors in [1] extend their work and use the computer game Quake to study the behavior of the game. 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 has been performed 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 provide a complementary solution and try to eliminate the global synchronization points and locks, i.e., making the game server “embarrassingly parallel” which aims at increasing the number of concurrent users per machine.

7. CONCLUSION

In this paper, we have shown that 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 thread pool 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. Running our prototype game, we show that the 8-core server can handle a factor of 2 more clients before the response time becomes unacceptable.

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 multi-threaded implementations. Another direction this work could be extended is to go beyond the single shared memory computer used and distribute the workload across clusters of computers. This could be achieved by implementing cross-server communication directly in the server code, or by using existing technology that makes cluster behave like shared memory machines. Furthermore, all experiments described here were run with an update frequency of 10 Hz. 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.

8. REFERENCES

[1] 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.

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

[3] 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.

[4] K.-T. Chen and C.-L. Lei. Network game design: hints and implications of player interaction. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, NetGames ’06, New York, NY, USA, 2006. ACM.

[5] H. S. Chu. Building a simple yet powerful mmo game architecture, 2008.

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

[7] B. Drain. Eve evolved: Eve online’s server model, 2008.

[8] F. Glinka, A. Ploß, J. Müller-lden, and S. Gorlatch. Rtf: a real-time framework for developing scalable multiplayer online games. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, NetGames ’07, pages 81–86, New York, NY, USA, 2007. ACM.

[9] J. Müller and S. Gorlatch. Enhancing online computer games for grids. In V. Malyshkin, editor, *Parallel Computing Technologies*, volume 4671 of *Lecture Notes in Computer Science*, pages 80–95. Springer Berlin / Heidelberg, 2007.

[10] K. Raaen, H. Espeland, H. K. Stensland, A. Petlund, P. Halvorsen, and C. Griwodz. A demonstration of a lockless, relaxed atomicity state parallel game server (lears). In *NETGAMES*, pages 1–3. IEEE, 2011.

[11] J. Waldo. Scaling in games and virtual worlds. *Commun. ACM*, 51:38–44, Aug. 2008.

[12] W. White, A. Demers, C. Koch, J. Gehrke, and R. Rajagopalan. Scaling games to epic proportions. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD ’07, pages 31–42, New York, NY, USA, 2007. ACM.