Project Lightspeed

Multithreaded simulation as applicable to multiplayer game servers

Ben Yuan, Yuchen Lin, Suzannah Fraker, Ben Cosman, Matt Dughi
Department of Computer Science
California Institute of Technology
1200 E California Blvd
Pasadena, CA 91126
{byuan, liny, sfraker, bcosman, mdughi} @caltech.edu

ABSTRACT

In the typical networked computer game, one or more clients communicate with one server process. The server process is responsible for performing the simulation required by the game and sending any results back to the clients, and the client processes are responsible for displaying the game state to their users and retrieving command input to send back to the server. Most games capable of supporting several thousand concurrent users choose to partition or duplicate the game universe in order to spread out server load accross independent server processes, but these solutions can be inconvenient for players and can still become overloaded. We consider mechanisms by which arbitrarily many players could be supported by a single server cluster that can dynamically provision computational resources as required by system load, and we implement a simple networked game as a proof-of-concept of an arbitrarily scalable game server software.

Categories and Subject Descriptors

C.2.4 [**Computer-communication networks**]: Distributed Systems—*Client/server*

General Terms

Distributed Systems

1. INTRODUCTION

The modern Internet era has brought with it the "massively multiplayer" genre of games supporting thousands of concurrent users in the same universe. It is a challenge to write server software for such games, as the requirements are steep: any server must be able to process input from a large amount of users, simulate the results of every interaction performed by every user's avatar, and send the results back to every user in real time - for any conceivable number of users. Evidently, there are limits to how many users a single node can process: after a certain amount of load is placed on the server, it simply becomes impossible for the node to support any more users while still running the simulation at full speed. Hence, it is inefficient to run an entire game world on a single node, unless the size of the game world is appropriate for the effective user limit.

For games where the expected user base is much larger than the user limit of a single node, there are a number of existing solutions. One common model is to run multiple copies of the game world, called "shards". Each shard is mutually independent of the others; there are usually few mechanisms by which users on different shards can interact with each other, and often player characters are not permitted to move between shards (without, for example, paying a "character transfer fee"). Another common model is to divide up the game world into multiple independent "zones" between which players can move at will, usually by traversing a designated boundary within the game world, and host each zone on its own node. This model works well when players are fairly uniformly distributed; however, network effects can make certain zones (like trade hubs) particularly popular, causing those nodes to be frequently congested at peak hours. In addition, the act of zoning carries a nontrivial computational cost, and objects on different sides of a zone boundary are usually invisible to each other, which places constraints on where zone boundaries can be located: it would be unacceptable if a zone boundary were located, for example, in the middle of a frequented public space.

This discussion applies primarily to games where real-time positional data is involved - in other words, for systems where a soft-real-time guarantee applies. Turn-based games, tick-based games (with tick intervals on the order of minutes or hours), and thinly disguised Web applications have different scalability issues that are not discussed here.

2. A CASE STUDY: EVE ONLINE

EVE Online is probably the best-known modern example of a single-shard universe, supporting tens of thousands of concurrent users in a single world. The world of EVE is naturally partitioned into 7929 distinct solar systems. EVE's internal architecture allows one or more solar systems to be placed on a single compute process [5]. With many compute processes distributed over around 60 physical machines (as of 2007), the entire game universe can be simulated.

However, each compute process can only handle so much load, and with EVE Online's gameplay structure capable of spontaneously generating load spikes of several thousand users at a time (for example, during fleet fights between major alliances), it remains a challenge to make sure that the game experience remains smooth. In the past, large fleet fights have been rough for the server cluster to handle, with long load times, long delays for actions, disconnections, and node crashes being common. A recent feature called "time dilation", which slows down simulation speed on heavily loaded nodes, has permitted large battles on the scale of 3,000 concurrent users [1] to proceed without being affected by the problems of the past, but slowing down simulation speed for a portion of the game world can have significant ramifications of its own. In particular, doing so gives both attacker and defender much more effective time to raise reinforcements, further compounding the load problem.

There are some solar systems that are continuously loaded - for example, Jita, currently the game's most important trade hub, which continuously averages over 1,000 concurrent users and processes tens of thousands of transactions per day. The Jita system is hosted on a dedicated physical server, but even so, the occupancy of the solar system must be capped at a particular level (currently around 2,300) to prevent the node from becoming overloaded and to maintain the high responsiveness required by a major trade hub.

The motivation for a dynamic scaling mechanism at the per-solar-system level is not hard to see. For continuously loaded server processes, it would enable essentially arbitrary capacity growth, allowing more traffic at peak hours. For the dynamic and unpredictable loads generated by fleet fights, it would enable additional capacity to be allocated on a whim, allowing such load to be absorbed easily.

3. RELATED WORK

Prior research by Müller and Gorlatch [2] has addressed the problem of distributing simulation load for a single node across multiple servers. Their approach, aptly called "multiserver replication", replicates the game state for a node across multiple proxy servers, each one responsible only for a subset of the actual computation. Each proxy server, after computing its own "share", rebroadcasts its own intermediate result to every other proxy, while simultaneously reading in intermediate results from every other proxy to construct a picture of the full game state that can be sent to clients. Their research addresses the problem of proxy failure and data redistribution, which is essential for a robust system. We considered using their architecture for our project but they do not appear to address the problem of provisioning additional proxy servers in response to load, which we wish to address. In addition, though this would allow a fairly conventional design for individual nodes, adding new nodes could potentially be a costly operation, as any new node would have to receive a full copy of the entire game state.

Other prior research by Raaen et al [3] addresses the problem differently, using a model they call a "lockless relaxed-atomicity parallel game server" (LEARS). In this model, every entity or action in the game world is assignable to a compute thread, and the thread pool updates every entity or action based on a definable schedule. This approach requires that the game architecture be specifically designed for parallelism - for instance, objects are required to use message-passing to change other objects - but rescaling becomes almost trivial: the thread pool need only be resized to deal with transient load.

One potential item of concern is that any parallel architecture where every node may potentially interact with every other node leads to quadratic scaling problems as nodes get added, which means that capacity scales less-than-linearly with increasing node count. This problem could be alleviated by traditional zone-partitioning, with dynamic scaling applied only to individual zones. This architecture would allow load transients like those in the EVE Online example (discussed below) to be absorbed, while still allowing a large single-shard universe without the quadratic overhead required by a single-zone world.

An attractive feature of the LEARS model is that it does away with most of the locking and other synchronization present in most models. The thousands of players playing the massively multiplayer games in question are in different locations, using different hardware, communicating over variously patchy internet connections, etc., and so given how coarse the server's knowledge is about which client commands were actually issued in what order, expending system resources ensuring that commands remain exactly in order is almost silly [3]. Instead, LEARS introduces a model where state is kept consistent by allowing objects to modify only their own state, while all other interactions are scheduled as "tasklets" to be executed by worker threads.

4. OUR PROJECT

We opted for the LEARS "tasklets" model for this project, largely because we were interested in exploring the practicality of the relaxed-synchronization model presented in that paper. This required that the game architecture be carefully designed to work within the constraints imposed by the "tasklet" model, but adding more worker nodes becomes a relatively simple matter.

The scope of this project was strictly proof-of-concept. We created a multithreaded, asynchronous, task-based simulation engine, as well as some example objects for the simulation. We present the result as a technology demonstrator in two parts - a server component and a client component - to validate the practicality of our chosen approach.

4.1 Server Architecture

The architecture for a single "node replacement", i.e. a server entity responsible for simulating all of the objects and interactions for a particular defined subset of the game world, consists of the following components:

- *One "master" thread.* This thread is responsible for keeping track of pending "tasks" and scheduling tasks for consumption by local worker threads. It also maintains a handle on all game objects for this node.

- *Several game objects.* Each game object represents an item in the game whose state might change. Changing a game object's state occurs only when some thread

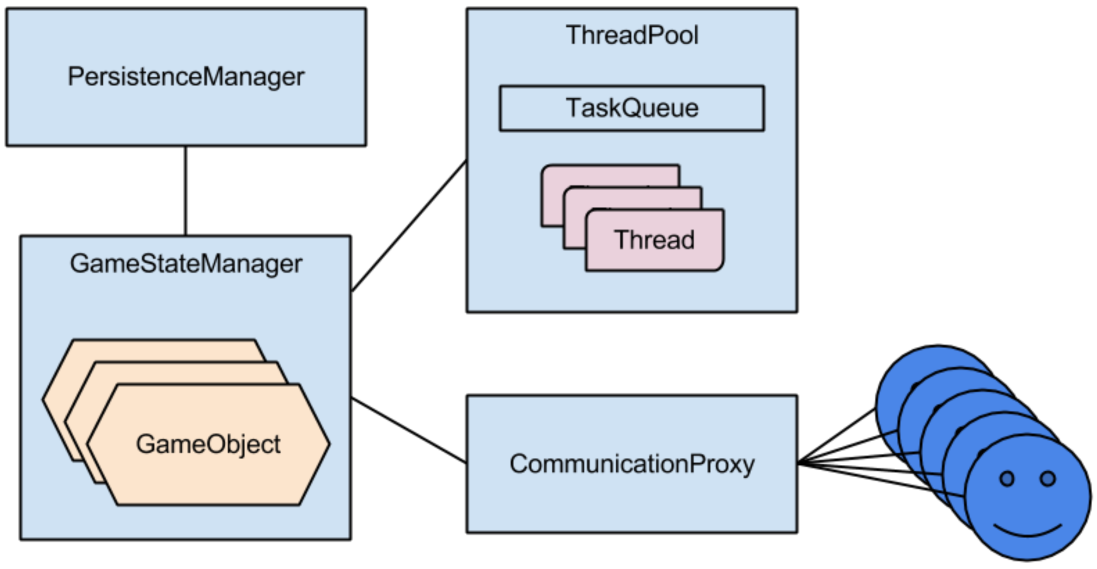

Figure 1: A high-level overview of our server architecture.

runs a task associated with that object. With proper scheduling of tasks, it should not be necessary to maintain atomicity of game object state changes.

- *Several tasks.* Almost every task is associated with an object and represents some interaction or update that object must process. A few other tasks (such as universal collision detection, which notifies two objects that they are too near each other) run without associated objects.

- *A few "worker" threads in a threadpool.* Each thread consumes a task from the master thread's ready queue, performs necessary calculations, and updates the state of the objects the tasks represent – potentially generating other tasks during this process.

- *A communication proxy.* This proxy spends most of its time idle, but it wakes up periodically to publish the current game state to all clients and to respond to game commands issued by clients.

- *A MySQL database.* The master thread interfaces with a database to load and save game objects and environment state to and from long-term persistence. This would be used when users log in, sign out, or open saved items.

Our technology demonstrator server runs one such node. The world is initialized with a planet and a field of asteroids with randomized initial positions and velocities. Once initialized, each object in the world schedules an update task every 100 ms. Additionally, every 100 ms, the system schedules a collision detection task, which identifies all pairs of objects that are sufficiently close to each other and applies a repulsive force to objects determined to be overlapping. Finally, every 200 ms, the server broadcasts a snapshot of the current game state to all connected clients.

Tasks are held in a 'waiting' queue until the server clock has advanced to the point where the task must be executed, then moved to a 'ready' queue from which worker threads can consume them immediately. An estimate of instantaneous load is taken approximately every 1000 ms by counting the number of threads actively processing tasks and adding the number of tasks in the ready queue; if this number is greater than the number of running threads at any point, the server is 'overloaded', i.e. it cannot keep up with the volume of tasks being scheduled. This state manifests as lag: tasks are executed later than scheduled.

4.2 Client Rendering

By way of comparison, the client compute architecture is quite simple, with most of the effort devoted to visualization. Here are its components:

- *A master thread.* Once again, there is a master compute thread. This thread is devoted to graphical visualization of interpolated game object actions, and currently simulates the server. (When interpolating the entire game state becomes too difficult, we will restrict this thread to interpolate only visible or local objects.)

- *Several game objects.* Exactly as in the server. In fact, the game objects in the client should be a (partial) mirror of all game objects in the server.

- *A client-side proxy.* This proxy also spends most of its time idle, but it wakes up whenever it receives an update of the game state or a command to forward to the server.

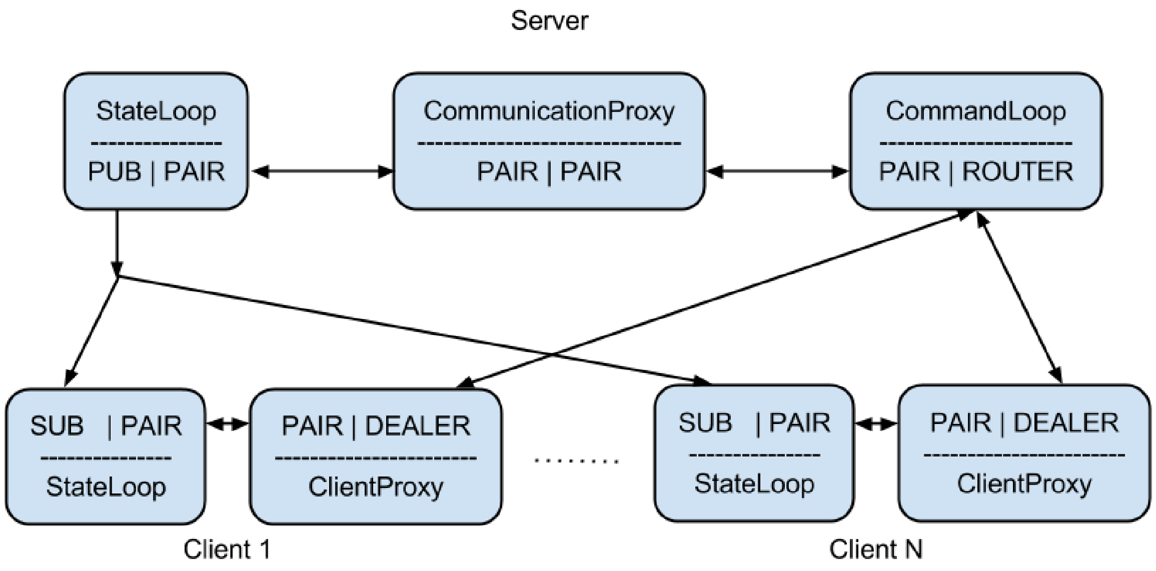

Figure 2: A graph of our client/server communication network, detailing threads and socket types.

The technology demonstrator client provides a visualization of the current server state, provided by 3D graphics engine OGRE using the library OpenGL. Because server state is sent only intermittently, and because of the potential for network outages, the client does a limited simulation of the visible game state in order to interpolate between state updates. This interpolation currently consists of running the simulation using the same update rules as the server, but with much smaller timesteps. This approach can cause problems with object warping during state updates, since some of the forces affecting objects are velocity-dependent. As a first iteration, though, the approach seems sufficient.

Because none of us are artists, all art assets have been procedurally generated. The background starfield was pregenerated by an external program called Spacescape. Asteroids are created by running the 'marching cubes' isosurface algorithm on a noisy three-dimensional scalar field weighted by distance. Planet heightmaps are generated by sampling a noisy three-dimensional scalar field along the surface of a sphere; planet textures are created by mapping heights to colors. Coherent noise is provided by the Perlin noise algorithm; the unique identifier for each procedurally generated object is used as a "seed value". Hence, every procedurally generated asset is different from every other one, but objects look the same across clients and accross separate runs of the technology demonstrator since the same unique identifiers are used on each initialization.

4.3 Client/Server Communication

Communication between the server and clients is mediated by proxy threads using the concurrency library ∅MQ as a transport layer. There are two types of messages that need to be transmitted between a server and all of its connected clients. Each message type is handled in its own thread and uses its own private communication port.

- *Game state from the server.* Game state messages consist of lists of objects in the zone the server process is currently tracking.
- *Game commands from each client.* Command messages are simply specifications of tasks the server should schedule immediately.

Our server and client proxies use several types of ∅MQ sockets to structure the flow of messages. All communications between server and client occur through TCP ports.

- State update threads follow a publisher/subscriber protocol (PUB/SUB, in Figure 2). In this protocol the server uses a single socket to communicate with arbitrarily many clients (which can join or leave the network at will). The server periodically pushes messages out to a port, and all connected clients simultaneously receive the same messages by listening to that port.
- Command threads use a somewhat more complex protocol (DEALER/ROUTER, in Figure 2) than state threads. The server once again uses a single socket to communicate with transient clients, but this time it is able to receive and send messages to specific clients. In our code, each client asynchronously sends an arbitrary number of commands to the server port, either requesting private data or sending tasks for the server to perform. The server then fair-queues these messages (while tracking the source address of each message) and schedules the relevant tasks using the game

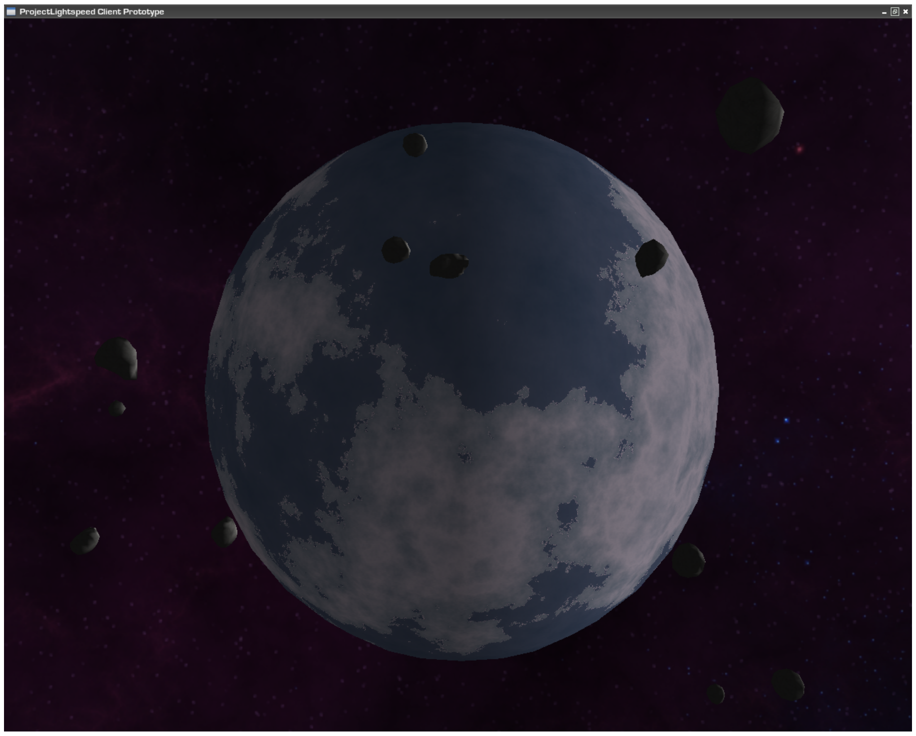

Figure 3: A screenshot of the technology demonstrator client rendering a planet and asteroids.

manager. If a client had requested a reply, then the server looks up the source address and sends a message only to that specific client.

- Finally, all our threads communicate with their parent threads using a pair protocol (PAIR/PAIR, in Figure 2). These are one-to-one asynchronous channels for miscellaneous communications such as passing data between the command and game state threads or broadcasting kill signals so that the child threads can join. As both socket ends of a pair occur in the same process (albeit different thread), they send messages over an interthread protocol instead of TCP, which is slower.

Encoding all parts of the application as strings was necessary for server-client communication. Data were serialized into strings by interpreting each byte as a character. At the other end of the communication channel, data were restored by reinterpreting an appropriate number of characters depending on the data type. This is obviously not endian-independent, or safe in any other sense for that matter, and would have to be refactored before a public release of the game. As it is, when an object is serialized, a string is built up of the byte strings representing each of its fields by concatenating them together. Since concatenation and truncation of strings is more efficient at the end, it was necessary to both add and remove from the end of the string (treating a serialized string as a stack). Thus, for each class, the fields were serialized in one order and unserialized in the reverse order. The first piece of data recovered from any serialized object string is information about the object's class, so that unserialization would not require any information outside of the string itself to reconstruct the correct objects.

4.4 Technology Demonstrator Behavior

Our demonstration consists of an asteroid field that users can interact with. Users can select a single asteroid and then shift-click a second asteroid to "throw" the first one at the second. We decided to implement limited physics unconstrained by reality, so asteroids bounce off each other quite energetically when they collide, and in free space have a noticeable drag force slowing them down. The user has considerable freedom to pan, zoom, and shift the camera view of the procedurally generated asteroids and planet against a starry background (see Figure 3).

5. OBSERVATIONS

The task scheduling system we wrote, because of the necessary consistency checks and context switching, incurs a small but significant amount of overhead for every task—in particular, large compared to simple tasks like physics updates. We observe that, when tasks are small compared to scheduling overhead, adding more threads is of limited utility. In particular, when the server is tracking as many objects as a single thread is capable of processing without significant load increases—on the test server platform (2.67 GHz Core i7 920, with ample available memory), we empirically determined this threshold to be around 11,000 objects—adding more threads causes the size of the ready queue (and hence the load estimator) to increase, from much less than 10 to more than 600 on the test platform.

On the other hand, when tasks are large compared to scheduling overhead, multithreading on a single machine does provide an approximately linear increase in load capacity. To verify this observation, we inject a small amount of synthetic load to each update task, forcing a sleep for 5 milliseconds.

A single thread can then only handle 20 objects without falling behind; however, adding more threads increases this threshold, up to 80 objects with 4 threads. We did not add more than 4 threads in any given run, since doing so would be of limited utility on a quad-core machine.

We wrote our task framework to use absolute timestamps everywhere. During testing, we observed that, from some machines, there were significant delays between task transmission and task execution. We determined that the delay was due to client and server clocks being out of synchronization. This problem could be remedied by using relative timestamps everywhere except the actual scheduler, but was not considered important for the technology demonstrator.

We successfully ran client applications on four machines simultaneously, with the server running on a fifth machine. Effects of commands from each of the clients were visible in all other clients. All of these machines had 64-bit architectures. We found that, while both the client and server applications could be compiled and run together on 32-bit machines, we were unable to use a client on a 32-bit machine with the server running on the 64-bit machine. This may be a result of the serialization paradigm, which depends heavily on the actual binary representation of data in memory.

6. FUTURE WORK

While the technology demonstrator provides a promising initial implementation of the ideas behind this project, it is substantially incomplete as a computer game, and indeed there are fundamental issues that we were not able to address in the time allocated. We discuss items for potential improvement, iteration, and addition here.

6.1 Large-Scale Implementation

The architecture of the technology demonstrator server assumes only one machine, on which all components of the server run as one process with many threads. We asserted as part of the motivation for this project that loads can potentially climb to the point where no single server can keep up with the simulation in real time. In such a scenario, the logical step would be to distribute the simulation among multiple physical servers, allowing a potentially unlimited number of computation threads.

There are serious issues that must be tackled to allow such an architecture, and we were not able to address them in the time given. An important unanswered question is that of dispatching tasks to remote threads: how much data must be sent, and how much data must be sent back? In the model where all object descriptions reside in a single shared memory space, it suffices to pass around addresses of objects—but when this model no longer holds, memory addresses become meaningless. One solution is to pass object descriptions to remote computation threads, and return descriptions back to the "master" thread pool; another solution is to use a replication model similar to the Müller-Gorlatch model, having certain thread pools responsible for certain sets of objects and broadcasting state among each other.

If we were to pursue this objective, we would want to implement one or both of these methods, and evaluate the effects on performance. The distribution of compute threads to multiple servers increases communication latency, which could impact both performance and accuracy; we would want to quantify these effects and determine whether the approach would be useful in high-load conditions.

6.2 Database

Storing persistent data is essential for representing long-term character progression and is useful in the case of moving characters between "zones", so we would need to construct a database backend. We began implementing this in this project using MySQL, a popular and free relational database solution which is easy to interface with from C++ using the library mysql++. We created one table to store the common data fields of every object that inherits from the base class SpaceObject, and another specifically for the class Asteroid, where the additional information characterizing each Asteroid could be stored. We wrote a PersistenceManager module which can connect to this database remotely and save and load Asteroids, but since it was not an essential part of the technology we wanted to demonstrate, we did not complete and integrate this module.

Further work on the database could integrate Memcache, a distributed memory-based caching system. The game data would be stored in key-value format in a distributed cache that is built on the memcache. This, to some extent, would alleviate disk latency. A memcache layer can handle all the reads and writes while a separate queuing process would run the data into the MySQL database. It is possible to run the MySQL, memcache, and queuing process on different machines in order to lighten the load.

We could experiment with a hashing algorithm (perhaps a modulo selection algorithm) in order to distribute the keys uniformly across the memcache server clusters. This would be an constant-time operation and also provide a good distribution of keys without necessarily creating overcrowded hot spots.

We would also need to consider the problem of a database server failing. The standard solution for this problem is to have at least one failover server that remains synchronized to the master server and can take over if the master server encounters issues.

The nature of a massively multiplayer online game is that it will quickly spread throughout the social graph. In a production environment, we would need to quickly be able to respond accordingly by scaling the storage layer without interruption. Perhaps we could accomplish this by simply doubling the size of the cluster by promoting the slave server to masters and adding the newly promoted nodes to the hashing, further removing items that no longer hash to the old master servers. This would give us flexibility and would not interrupt the application layer. Or perhaps we would take advantage of another concept called vbuckets [4].

6.3 Bandwidth Reduction

In our technology demonstrator, game state changes are currently encoded by serializing all game objects and publishing this full state. In order to reduce bandwidth, a planned extension would be to frequently publish differential ("diff") updates instead of full game updates. These diff updates would only encode the game object fields that have been modified since the previous diff update. Our network topology already supports handling such messages, but we still need to implement time-stamping of changes so that the server proxy can know which fields it needs to send.

If diff updates are ultimately implemented, we would need a procedure for providing new clients a copy of the full game state. This could trivially be accomplished by querying the server through the command thread and forwarding the server's reply (a full game update) to the state thread. It would be very natural to integrate private full game updates with login/authentication requests (another possible extension).

7. CONCLUSIONS

Our testing suggests that, as long as scheduling overhead can be kept small compared to task size, the architecture we have demonstrated could be very useful in creating arbitrarily scalable game server software.

8. ACKNOWLEDGMENTS

The authors would like to thank Professor Steven Low for his guidance on this project. All funding was provided by the Computer Science department of the California Institute of Technology.

9. REFERENCES

[1] E. Grabianowski. How the Battle of Asakai became one of the largest space battles in video game history. http://io9.com/5980387/how-the-battle-of-asakai-became-one-of-the-largest-space-battles-in-video-game-history.

[2] J. Müller and S. Gorlatch. Rokkatan: Scaling an RTS game design to the massively multiplayer realm. *ACM Computers in Entertainment*, 4(3), July 2006.

[3] K. Raaen, H. Espeland, H. K. Stensland, A. Petlund, P. Halvorsen, and C. Griwodz. LEARS: A lockless, relaxed-atomicity state model for parallel execution of a game server partition. In *The 41st International Conference on Parallel Processing Workshops*, 2012.

[4] Scaling memcached with vBuckets. http://dustin.github.io/2010/06/29/memcached-vbuckets.html.

[5] Tranquility - EVElopedia. http://wiki.eveonline.com/en/wiki/Tranquility.