A Distributed Architecture for MMORPG

Marios Assiotis and Velin Tzanov
Massachusetts Institute of Technology
{assiotis,tzanov}@alum.mit.edu

ABSTRACT

We present an approach to support Massively Multiplayer Online Role-Playing Games. Our proposed solution begins by splitting the large virtual world into smaller regions, each region handled by a different server. We present techniques and algorithms that (1) reduce the bandwidth requirements for both game servers and clients, (2) address consistency, hotspot, congestion and server failure problems typically found in MMORPG and (3) allow seamless interaction between players residing on areas handled by different servers. By implementing a simple game, *Kosmos*, we show the applicability of our approach as well as the relative performance benefits of designing new games using our architecture.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: COMPUTER-COMMUNICATION NETWORKS—*Distributed Systems*

General Terms

distributed architecture

Keywords

Multiplayer Game, MMORPG

1. INTRODUCTION

In this paper we propose an architecture for Massively Multiplayer Online Role-Playing Games (MMORPG) to support a very large number of concurrent users. Our design allows for unrestrained growth of the virtual world while remaining practical and pragmatic with respect to how such games are implemented today. Our architecture is based on the fact that MMORPG exhibit strong locality of interest and as such we can split the large virtual world into smaller regions. Multiple servers, still under the centralized control of the game publisher, are each assigned to handle such a region. A static division, however, will not be able to react to sudden load peaks caused by so called hotspots. Our design allows for a reorganization of the division without interrupting gameplay significantly. We also propose techniques and algorithms to handle player interactions between regions handled by different servers.

Netgames'06, October 30–31, 2006, Singapore.

We begin in section 2 by defining a formal model used throughout this paper to describe our design. In section 3 we describe our design, propose a solution for dealing with events near server region boundaries as well as algorithms for seamlessly transfering objects between servers without interrupting gameplay. Section 4 details algorithms build on top of our design, enabling dynamic reorganization of the division of regions amongst servers as well as the introduction of new servers to allow for virtually unlimited scalability. Finally, in section 5, we list our emperical and experimental results and confirm that the proposed architecture can achieve practical performance, scalability and seamless gameplay.

1.1 Related Work

There are various ongoing research efforts in developing scalable architectures for MMORPG. Most research trends are towards peer to peer (P2P) systems[1, 5, 6]. P2P systems exploit the locality of interest feature in MMORPG - much like our proposed architecture. Although such systems frequently describe novel mechanisms for distributing load and scaling effectively, they are not very pragmatic in a real world commercial setting. P2P systems are not under the centralized control of the game publisher; the game state is stored in the clients, with each client being responsible for a smaller region. Clients multicast updates to other peers. However, lack of an established IP Multicast solution, forces such architectures to consume a lot of bandwidth. Reliance on nodes with high latency network connections for data transfers also means that it becomes increasingly difficult to handle scenarios where players are interacting near areas handled by different nodes. In addition, P2P systems are less secure when compared to a centralised solution. In P2P systems the global game state is usually stored in the local client. Consequently a malicious player could modify the game state.

In [8], each game server manages several dynamically assigned *microcells*, each of which contains a very small portion of the large virtual world. The Microcells can be rearranged between servers to balance game load efficiently. The authors do account for interaction near the microcell borders and propose various methods for efficiently balancing the load. In order however to minimize the overhead incurred by inter-cell interactions, microcells depend on a

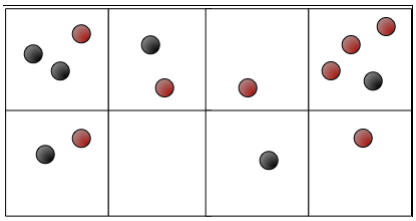

Figure 1: ***The squares represent different servers each one handling a separate area of the virtual world. The circles represent players and NPCs.***

single shared storage mechanism. A shared storage mechanism is a scalability bottleneck and could potentially suffer from a large performance penalty as the game world grows.

In a mirrored game architecture[3], the clients are balanced across a large number of servers. Each server holds an identical copy of the game world. Unfortunately, it becomes very hard to maintain consistency among all servers in such a highly variable environment, with thousands of concurrent players. In addition to that, each single server may not have the processing power to evolve the entire world(Game AI).

2. DEFINITIONS

We proceed to describe a formal model used throughout this paper. We define a game character controlled by a human player as a *player* and a character controlled by AI as *NPC* (Non-Player Character). Now consider the virtual world W which can be modeled as a 2D geographical map, although, it can easily be extended to 3D. Players, NPCs and items in the virtual environment are all considered game objects. Every object $i \in W$ has a set of coordinates $C_i(x, y)$ on the map as well as state S_i, which we treat simply as a set of bits. Every animated object's trajectory function $f_i \subset S_i$ describes the objects current movement in the world with respect to time. Throughout the paper we refer to the software running locally on each player's computer as *clients*.

2.1 Events

Every object follows its trajectory unless a game event occurs. An event is an atomic transaction that happens in the world and changes one or more objects' state. Events are discrete and of zero time duration. We treat non-instantaneous operations as a set E of events with $|E| > 1$ and where necessary, encapsulate movement in trajectories. As an example, consider the firing of a rocket. In our design, the entire operation is not a single event but two events: (1) the initial firing of the rocket along with the creation of the rocket object and (2) the final impact and the destruction of the rocket object as well as any other objects affected by the impact. The path travelled by the rocket is encapsulated in its trajectory.
There are two types of events in the system : (1) events caused by input from the human player and (2) events caused by game AI rules. Also, it is paramount to note that events can act on objects only within a fixed range R - where R is the largest value amongst all event radii and player sensory capabilities.

2.2 Actions

A large number of events in the game are caused by human player input. We say that human player input constitutes an *action* which in turn creates an event. For example, if a player wishes to execute an action, such as firing a rocket, the client sends the action to the server to which it is connected. The server creates an *event* and updates the state of all affected objects as necessary. The client is then notified of these state changes and updates its local copy.
Each action sent from the client to the server carries a monotonically increasing ID number. As we will examine later, ID numbers are used in some scenarios to ensure correctness and thus maintain consistency.

2.3 Area of Interest

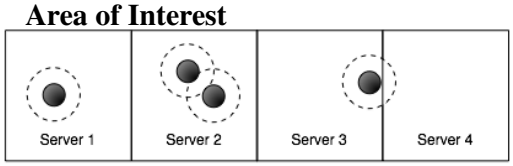

Figure 2: ***The concept of an area of interest with multiple servers. From left to right, the first player's area of interest lies entirely within the first server. The second and third players also lie entirely inside the second server. The fourth player's area of interest spans across server 3 and server 4.***

It follows naturally from the nature of MMORPG that players are only interested in events occuring within the player's sensory capabilities. As discussed in [6] a client does not need to receive events that the player cannot see or hear. We can therefore define an *area of interest* for a player to be the set of all points at most R away from the player's location.

3. DESIGN

The overall system architecture is based upon the spatial locality of interest players exhibit. Clients can be clustered together depending on their player's location in the virtual world. Therefore the virtual world W can be divided into smaller, *disjoint* regions $w_1, w_2..., w_n \subset W$ - with each region being assigned to a different server as shown in figure 1. The region assigned to each server can be any convex polygon. The multiple, smaller regions are transparent to the player, who only sees one big virtual world.
At all times, the client has only one main point of contact, the server to which it sends actions to. For each client, the server acting as the main point of contact is determined by the location of the player on the virtual map. The client continues to send actions to the same server, unless it is notified otherwise by the server[1]. Even though clients send actions to only one server, they may receive events from multiple servers. This is the case when players are located at a distance less than R from another region.

The system infrastructure consists of multiple servers, all interconnected with a mean latency of L_S for server to server network communication. Let L_P be the mean client to server latency and assume $L_P \gg L_S$. The latency assumption is justified as clients typically run on commodity hard-

[1]We examine under which circumstances clients change servers in section 3.3.3

ware and are connected to the internet using broadband or dialup connections. Game publishers usually install game servers in the same datacenter or interconnect them using high speed, low-latency backbones. We also assume the existence of a *lobby server* which handles user logins and initially informs clients which server to connect to.

To summarize, in section 3.1 we examine a locking mechanism used to achieve consistency when processing events across multiple servers. Then in section 3.2 we describe how our design augments traditional publish/subscribe systems and in section 3.3 we use our locking and event mechanisms to propose various algorithms for dealing with events that span across regions handled by more than one server.

3.1 Locking Mechanism

Region Locks

To tackle the various consistency challenges that arise, we introduce the concept of region locks. Region locks are locks over geographical areas in the virtual world. The authority for granting region locks lies entirely in the server handling the region. A server executing an event affecting a specific area on the map (e.g. a bomb explosion), might request a lock over that area. Once a server is granted a lock, other servers requesting a lock for an area that overlaps with the already locked area, would have to wait in a queue.

Object Locks

When processing events near server region boundaries, in addition to the region-based locks above, it may be necessary to obtain locks on the objects affected. When a server obtains a lock on an object, another server requiring the same object for the execution of an event, will have to wait until the first server releases the lock.
Once again, the authority for granting locks is with the server owning the object, that is the server handling the object's state.

3.2 Event Announcing Mechanism

In this section we describe a mechanism for servers to announce events to clients and other servers by using a publish/subscribe design pattern[4, 2]. In our design servers are both publishers and subscribers whereas clients are only subscribers. Subscription is region-based; a subscriber may subscribe to receive events occuring only within a small region.

We first examine how servers announce events to other servers. Every server is subscribed to its adjacent servers for all points R distance away from its boundaries. For example, assume a virtual world handled by two servers, S_1 and S_2, such that S_1 handles the right half of the area and S_2 handles the left half. In our system, S_1 will be subscribed to S_2's rightmost area and S_2 will be subscribed for events occuring within S_1's leftmost area, always within a range R. As such, both servers will always notify each other for events occuring within R of their borders. Having servers subscribe to each other for events occuring within distance R of their common border, is an important part of our design. In the next paragraph, we show how we utilize cross-server subscription to allow for smooth gameplay in areas near server borders.

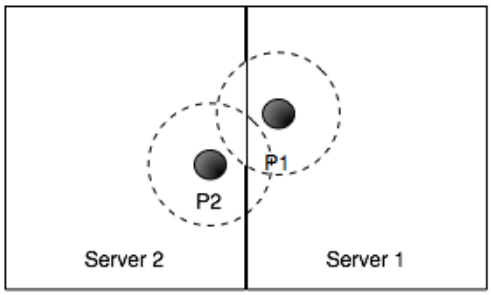

Figure 3: ***An example of two adjacent game servers and two clients*** P_1 ***and*** P_2 ***each inside an area handled by a different server***

We now examine how servers announce events to clients. For each player in a server's region, the server performs two important functions : (1) it processes actions received from the client and (2) it subscribes/unsubscribes the client to receive events occuring in their *area of interest*. In a single server scenario, the server would subscribe and unsubscribe the client as the player moves through the virtual world so that the client only receives events relevant to the area of radius R from the player's coordinates. For example, a player P_1 can walk in another player's P_2 area of interest. The server receives the player walk action from P_1 and announces the event that P_1 walked to both P_1 and P_2. If both players are looking in the right direction, they should now be able to visually spot each other.
In a multiple server scenario, whenever a player's coordinates are at most distance R from a point in another server's region, his *area of interest* may span across more than one server, and as such the client should receive events from all relevant servers. A trivial example, shown in figure 2, would be a player walking in an area handled by server S_3 towards an area handled by server S_4. As soon as the player is within distance R from S_4's border, and because S_4 is subscribed to receive events from S_3, S_4 will learn of the player's presence. S_4 will then automatically subscribe the player's client to receive events happening within the portion of the player's area of interest handled by itself. We follow the same procedure in reverse for unsubscribing clients. If the player starts walking away from S_4, then S_4 will subscribe the client to receive events occuring in a smaller and smaller area inside S_4 until the player is more than R away from S_4's border. Once that happens, S_4 will receive an event from S_3 that the player has moved and S_4 will completely unsubscribe the client. Note that even though a player can be subscribed to receive events from multiple servers, it only sends actions to one server - in our example from figure 2 above, P_4 only sends actions to S_3.

3.3 Events Near Server Boundaries

One of the major advantages of our design is that it handles interactions near and across multiple servers well. We examine algorithms to handle such complex scenarios; we also provide the consistency requirements necessary to ensure correctness.

3.3.1 Consistency Requirements

To maintain correctness, two consistency requirements must be satisfied at all times

1. The order of events affecting the state of any given object must be the same for all clients.
2. There exists a global order of the events that is consistent with the order of events acted on every object. That is to say it should be possible to assign numbers to all events, so that the sequence of numbers of the events that happened to any one object is monotonically increasing.

3.3.2 Events

We now solve the problem of events that affect areas in multiple servers. To tackle this problem we use the concept of region locks introduced in section 3.1 and the event announcing mechanism from section 3.2. Assume an event originates on server S_2 but also affects areas on servers S_1 and S_3. Our algorithm performs the following

1. Server S_2 requests region locks for the geographical areas affected by the event and object locks for all objects participating in the event. To avoid deadlocks, begin requesting both region and object locks in decreasing ID order. In our example, S_2 asks and obtains all the locks from S_3 first, then obtains its own locks and then from S_1. Locks are requested from each server in an *all-or-nothing* approach - S_2 will request all necessary locks at each step and will not proceed to request locks from another server until all locks are available first.
2. Server S_2 executes the atomic event and notifies S_1 and S_3 how they should change their state.
3. S_2 releases all acquired locks
4. Clients receive all the state updates via the mechanism described in section 3.2
5. Once the client receives the outcome of the event from all the servers it is subscribed to, the client updates its local state and updates the human player's game view if necessary
6. If a client receives a new event before it receives the outcome of the previous event from all the servers, then the client queues the new event and executes it right after being notified about the previous event from all servers

In the case where the virtual world is divided into rectangles, the number of servers involved in the execution of an event is not more than 4. As such, the maximum additional delay incurred is $O(L_S)$.

3.3.3 Object Transfers

A special case of event occurs when an object moves from a region handled by one server to a region handled by a different server. Our solution for this case is very similar to the algorithm in section 3.3.2, however, executing the event includes transferring the state of the object from one server to the other. The most complex scenario occurs when the object is actually a player. In that case, not only do we need to transfer object state between servers, but also, events occurring while the transfer is in progress must be processed as normal. In more detail, consider the scenario of a player P, walking from S_1 to S_2. Our algorithm for server S_1 performs the following steps

1. S_1 requests region locks for the small area player P will walk through, beginning with the server having the largest ID. In this example, S_1 asks and obtains a lock from S_2 first before obtaining its own lock.
2. Obtain an object lock on P
3. S_1 sends the ID of the last action processed for player P to S_2[2]
4. S_1 initiates the transfer of P's state to S_2
5. Upon state transfer completion, S_1 releases previously acquired locks
6. S_1 notifies P that S_2 is now his new point of communication
7. If P initiates any events while steps 4, 5 and 6 are being executed, S_1 forwards them to S_2

Similarly, the destination server S_2 performs the following steps

1. Let S_1 acquire region locks
2. Receive the ID of the last player action processed by S_1 and store it as P_{lastID}
3. Accept the transfer of state
4. When state transfer completes, accept the connection from P
5. If S_2 receives an action from player P having $ID > P_{lastID} + 1$ queue it up. If $ID = P_{lastID} + 1$, process the action and exit the algorithm
6. At any point, if an action by P is forwarded to S_2 by S_1, process it and set P_{lastID} to $P_{lastID} + 1$
7. When the first event in the queue has an ID equal to $P_{lastID} + 1$, process all events in the queue and exit the algorithm

The event ID tracking mechanism ensures that the consistency requirements laid out in section 3.3.1 are met, even if some events forwarded from S_1 to S_2 are slightly delayed. In section 4.2, we will examine how we utilize this feature of our design to dynamically reorganize the virtual world division without affecting gameplay.

[2]Recall from section 2.2 that events caused by player actions have monotonically increasing ID numbers associated with them

From a player's prespective, everything continues smoothly. A client continues to send actions to S_1 unless notified otherwise in step 6 of the first algorithm. Our design for multiple publishers as described in section 3.2 ensures that clients continue to receive event notifications without interruption and with only an additional mean delay of L_S. As $L_S \ll L_P$, the delay is not noticable.
The algorithm works equally well for objects other than human players. In such cases, there is no need to notify the object of the server switch - the new server becomes responsible for applying game AI rules upon sucessful receipt of the object's state.

3.4 Aborting Events

Due to the consistency requirements or delays introduced during network transfers, it may be the case that the prerequisites for executing an event are no longer valid. For example, two players attempt to pick up the same object at approximately the same time instant. The server will obtain locks on behalf of the client whose action the server received first. The server will then execute the event and notify the first client that he has sucessfully picked up the object. After doing so, the server releases the locks and attempts to lock the same object on behalf of the second client. However, the object has been already picked up by the first player and naturally the server will not be able to acquire the locks. Therefore the event on behalf of the second client will be aborted and he will be notified that the object is no longer available.

4. LIVENESS AND SCALABILITY

In this section we discuss some important facets of MMORPG as a distributed system. We also introduce an algorithm for dynamically balancing the server and network load by reorganizing the game world while the game is in progress.

4.1 Liveness

Our design allows for a seamless player transfer between servers that does not require large data burst transfers. Also, our solution to congestion and hotspots, presented in the next section, does not create any sudden peaks in network traffic.
To show that our design has this desirable feature, it is enough to satisfy the following property : At any time instant, the data exchange between a server and a client as well as the data exchange between two servers is limited to $O(1)$. In other words, at no point in time is there a large game data exchange between servers or server and client. It is trivial to see that none of the steps in the algorithms from section 3.3.3 break this property.

4.2 Hotspots

To address unpredictable congestion and hotspots that typically occur in MMORPG, we describe an algorithm for splitting the area handled by one server into two or more parts. The congested parts can be assigned to either new or existing servers, depending on the current capacity of the system.

Consider the example for two servers, S_1 and S_2. S_1 is operational but experiencing a heavy load as it is responsible for a large number of objects and is handling an unusually high number of events. S_2 is not operational yet and is without any state. Our algorithm performs the following steps :

1. S_1 designates an area as belonging to S_2.
2. S_1 begins transferring game data to S_2. The game data consists of objects that can be serialized and marshalled across the wire. Only game data in the area designated as belonging to S_2 in step 1 is transferred.
3. If while the transfer is in place, an object already transferred is updated, S_1 sends the updated copy to S_2.
4. Upon data transfer completion, S_2 can immediately become operational as it has all the game data required, including object trajectories.
5. S_1 triggers the split and S_2 begins publishing events.
6. The last two steps from the player transfer algorithm detailed in section 3.3.3 are executed for every player that is now located in an area handled by S_2.

The key in understanding how our algorithm satisfies the liveness property is noting that the communication in steps 2 and 6 happens over an extended period of time. In step 2, S_1 can throttle the data communication between itself and S_2 depending on current inter-server network congestion. More importantly, in step 6, our algorithm from section 3.3.3 ensures that no events are lost. Therefore, even if S_1 notifies clients slowly and over time, that their main point of contact is now S_2, gameplay will not be significantly affected. The additional mean delay incurred while the transfer is in progress is L_S, where $L_S \ll L_P$. Therefore even for large N, the transition is seamless for players.

4.3 Scalability

As the number of players grows with the popularity of the system, it may be necessary for the game publisher to grow the virtual world. Below we describe a simple mechanism for joining a new game server S_{new} to the network.

1. S_{new} subscribes to its adjecent servers for all regions at most distance R from its boundaries
2. All servers adjecent to S_{new} subscribe to receive events occuring in regions at most distance R from their respective boundaries
3. S_{new} becomes operational

Adding new nodes can be an effective means to increase the size of the virtual world, in order to accomodate for an increasing user base.

4.4 Fault Tolerance

MMORPG users expect game servers to be constantly available with little or no downtimes. Recent outages in the *World of Warcraft* network have cost the game publisher, Blizzard Entertainment, an estimated $26198 per hour[7]. Our design, allows for a mirroring scheme that can be used to cope with server failure. Below, we provide a simple outline that can provide fault-tolerance under some common

failure scenarios.

One backup server S_b is assigned to each operational server S_o which acts as the primary server. We then use a methodology similar to the one used in the region-split algorithm in section 4.2. This time however, S_b subscribes to receive event updates in the entire geographical region handled by S_o. Thus S_o announces *all* events to S_b prior to announcing to anyone else. In addition, S_o informs S_b every time it acquires or releases a lock. Unlike the algorithm in section 4.2 however, we do not notify the players of S_b's presence unless we detect that S_o is down. In that case, S_b, having all the game data, executes step 5 of the region-split algorithm. The transfer of game data itself is $O(L_S)$. With an appropriate failure-detection scheme, the total delay between primary server failure and the backup server being fully operational can be as short as the maximum delay for server-to-server communication. The above mechanism is simple and showcases the flexibility of our design. Much more work needs to be done before we can claim a complete solution for coping with server failure. However, providing a thorough discussion on fault tolerance issues is beyond the scope of this paper.

5. EVALUATION

5.1 Implementation

We have implemented a simple game called Kosmos, to demonstrate the most important aspects of our architecture as well as measure the relative performance gains of the distributed approach. Kosmos is built on top of the Java Remote Method Invocation subsystem. Although this is inefficient from a performance perspective, it allows for a simple event-driven design that clearly demonstrates how our architecture works. We have also implemented a separate subsystem for facilitating asynchronous method invocations. Our purposes was to further test if our design can correctly support asynchronous operations, a mode of operation typical in networked games.

5.2 Empirical Results

We played a version of *Kosmos* on a local network of PCs using a GUI game client which allowed a human player to navigate in the virtual world and fire rockets using the mouse. The GUI also displayed a bright red line to show server region borders. As such we tried to manually create failure scenarios by simultaneous fire, constantly crossing the virtual server border back and forth, transferring objects between servers and simulating collisions exactly on server borders. Kosmos passed all tests successfully. Despite the lack of optimizations, the gameplay was very enjoyable.

5.3 Experimental Results

We performed our experiments using a version of *Kosmos* on a large heterogeneous cluster. To do so we used Emulab[9] to configure a network topology of multiple servers and clients. All game servers were connected to a high-speed LAN with virtually zero latency and zero packet loss rate. Game clients were configured with various latencies ranging from 10ms to 300ms and variable packet loss rates, ranging from 0 to 0.5%. The hardware configuration consisted of 3GHz Pentium IV PCs with 1GB of RAM running RedHat Linux 9.0 for the game servers and i586 variants for the game clients. To evaluate our experimental results we measured the time it

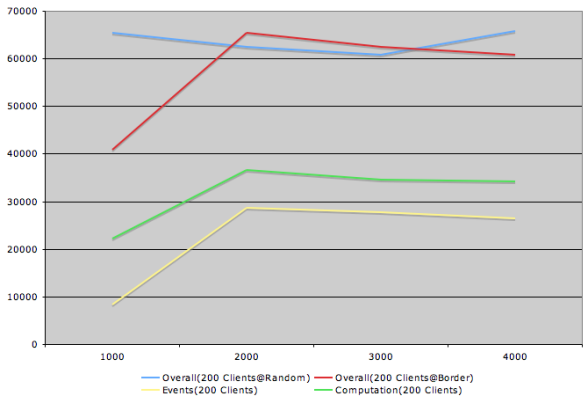

Figure 4: ***Testing performance when 200 clients are spawned at random locations in the virtual world and when spawned near server region boundaries. It can be seen that performance does not suffer even when all clients are spawned near server borders.***

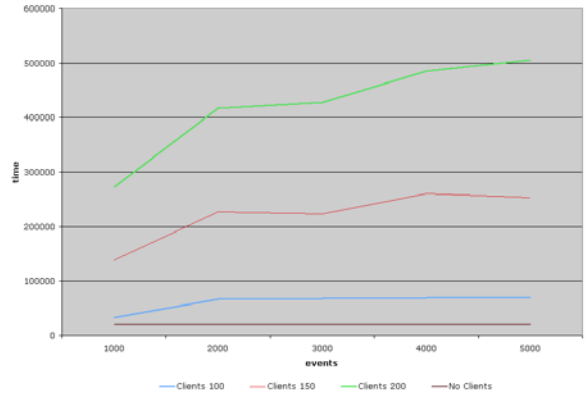

Figure 5: ***The performance of a single server with 100, 150 and 200 clients. We can see that the single server approach has great difficulty scaling with the number of users.***

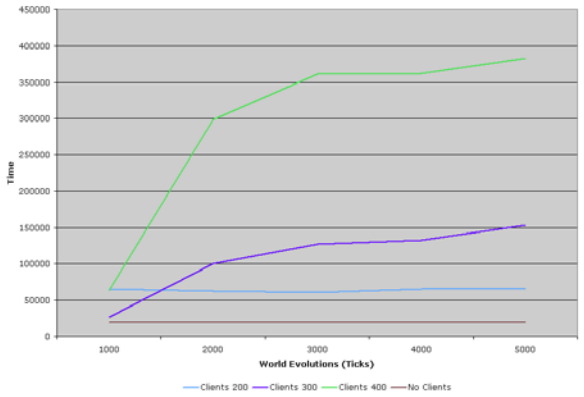

Figure 6: ***The performance of three servers using our architecture with 200, 300 and 400 clients. We can see that three servers handling 400 clients outperform a single server handling 200.***

takes to process events under various loads; starting from 1000 events and slowly increasing the number of events to 5000.

Experiment 1

We tested the practical performance of our algorithms for handling gameplay near server borders. To do so, we began by testing with 200 clients whose players have uniformly random coordinates and are thus placed everywhere in the virtual world and then comparing with 200 clients whose players all have coordinates within R of server borders. From our results, shown in figure 4, we conclude that despite the additional complexity, our design handles gameplay near server borders very well.

Experiment 2

To evaluate how well our architecture scales with respect to the number of users, we measured the performance of single and multiple server configurations. Figure 5 clearly demonstrates the practical limitations of a single server approach. Even though our event announcing mechanism described in section 3.2 allows a single server to scale gracefully up to 150 clients, server performance drop radically at 200 clients. Our architecture however, can scale to handle a large number of clients. Test results shown in figure 6 demonstrate that three servers can handle 400 clients. Figure 4 suggests that players do not notice the slight delays incured due to multiple servers.

6. CONCLUSIONS AND FUTURE WORK

In the future, further improvements are possible, especially in the area of fault-tolerance. In addition, this paper does not deal with security and authentication. However, given the fact that our architecture retains game state control in the hands of the game publisher, security and authentication could be incorporated in our architecture relatively easily.

Overall, our solution takes advantage of the locality of interest to distribute the game across several game servers and reduce both the computational strain as well as the bandwidth requirements on each one. Furthermore we present a solution to the problem of handling game events occurring near virtual boundaries, provide seamless transfer of objects between servers, describe an algorithm for dealing with hotspots, and discuss an algorithm that allows the entire game to scale horizontally. In conclusion, through experimentation we have shown that a new game written with our architecture can scale to handle a large number of players.

7. ACKNOWLEDGMENTS

We would like to thank Professor Robert Morris and Panayiotis Mavrommatis for their valuable feedback.

8. REFERENCES

[1] A. R. Bharambe, S. Rao, and S. Seshan. Mercury: a scalable publish-subscribe system for internet games. In *NETGAMES '02: Proceedings of the 1st workshop on Network and system support for games*, pages 3–9, New York, NY, USA, 2002. ACM Press.

[2] S. Caltagirone, M. Keys, B. Schlief, and M. J. Willshire. Architecture for a massively multiplayer online role playing game engine. *J. Comput. Small Coll.*, 18(2):105–116, 2002.

[3] E. Cronin, B. Filstrup, A. R. Kurc, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. In *NETGAMES '02: Proceedings of the 1st workshop on Network and system support for games*, pages 67–73, New York, NY, USA, 2002. ACM Press.

[4] S. Fiedler, M. Wallner, and M. Weber. A communication architecture for massive multiplayer games. In *NETGAMES '02: Proceedings of the 1st workshop on Network and system support for games*, pages 14–22, New York, NY, USA, 2002. ACM Press.

[5] T. Iimura, H. Hazeyama, and Y. Kadobayashi. Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 116–120, New York, NY, USA, 2004. ACM Press.

[6] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games, 2004.

[7] R. Miller. Extended Outages for World of Warcraft.

[8] B. D. Vleeschauwer, B. V. D. Bossche, T. Verdickt, F. D. Turck, B. Dhoedt, and P. Demeester. Dynamic microcell assignment for massively multiplayer online gaming. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[9] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.