A Distributed Architecture for Massive Multiplayer Online Role-Playing Games

Marios Assiotis Velin Tzanov

May 13, 2005

Abstract

We present an approach to support Massively Multiplayer Online Role-Playing Games using a centralized distributed architecture by splitting the large virtual world into smaller areas. Our approach takes significant advantage of the locality of interest such games exhibit to reduce the bandwidth requirements for both game servers and clients. We also propose a solution to the hard problem of interaction between players residing on areas handled by different servers. We have also implemented a simple game, *AlienBorder* which demonstrates the correctness of our approach as well as the relative performance benefits of writing new games using a distributed architecture versus a non-distributed one.

1 Introduction

In this paper we propose a centralized, distributed architecture for Massive Multiplayer Online Role-Playing Games (MMORPG), to support a large number of concurrent users, without sacrificing efficiency or security. The primary contribution of this paper is architectural. We take advantage of the locality of interest data exhibits in an MMORPG to separate the large world into smaller individual regions and assign each region to a different physical machine.

Our most important technical contribution is a design that handles game scenarios occurring in an area near the virtual border separating two or more servers. Existing systems solve this problem by using inner-game mechanisms such as water, tunnels or teleports. We present a novel way that allows for game event to execute normally in such areas while being completely transparent to the user and not increasing the latency between game client (player) and server significantly.

Although we do not discuss fault-tolerance, our design allows for a number of techniques to be applied. A short discussion about fault-tolerance is included in the end.

The rest of the paper is organized as follows: Section 2 presents some previous work done on MMORPG systems while section 3 provides background material for MMORPGs. Sections 4 and 5 present the challenges we faced and how our design solves the major issues identified. Section 6 discusses the implementation of AlienBorder, a simple game built using our architecture, used to demonstrate the feasibility of our design as well as evaluate the performance of it. Finally section 7 concludes with our results and discusses shortcomings and future work.

2 Related Work

In this section, a number of other approaches are discussed as well as the various problems each one has.

Client/Server

In this architecture a single server is responsible for holding the entire game state and han-

dling all clients. Obviously this can not handle the extremely big load required by MMORPG. Such an approach is typically used in *First Person Shooter* games such as *Quake* and *Doom*.

Mirrored Game Architecture

Similar to the client/server architecture, although the clients are balanced across a large number of servers. Each server holds an identical copy of the game world. Cronin et al[CFKJ01], discuss such an architecture as well as an efficient yet complex synchronization scheme[CFKJ02]. 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).

Peer-to-peer

A lot of current research trends are towards P2P systems[KLXH, BRS02], which although interesting are not pragmatic in a real-world commercial setting. P2P systems exploit the locality of interest feature in MMORPG, much like our proposed architecture. However, in P2P systems, the game state is stored in the clients, with each client being responsible for a small region. Clients multicast updates to other peers. However, lack of an established IP Multicast solution, forces such architectures to consume a lot of bandwidth. In addition, P2P systems do not currently have a reasonable way to handle cases were areas of interest overlap[1], other than use game mechanisms such as a tunnel or a teleport.

3 MMORPG

We now move to discuss a few key characteristics of MMORPG. Readers familiar with Role Playing Games can skip this section.

[1] the so-called "player on region border" scenario which we discuss extensively

3.1 Virtual World

MMORPG attempt to simulate real-life as much as possible. As such it is necessary to constantly evolve the game world using a set of laws. These laws are a complex set of rules that the game engine applies with every clock tick. A trivial example of such a rule is a player walking near an animal (NPC) and as a consequence of that action, the animal would attack the player.

The virtual world consists not only of human players but also of all game elements that are not living objects. These elements are immutable and include the area terrain, trees, mountains, rivers, etc.

3.2 Player

A player is defined as a single human player, that controls a single character represented by an avatar in the virtual world. Each player has a unique state consisting of several character properties. Such properties typically are health, abilities, belongings and other depending on the specifics of the game. In a typical game the player would take on missions or quests that require traveling in different parts of the virtual world.

3.3 Non-player characters

A large number of non-player characters (NPC for short) such as animals and even computer controlled opponents make their appearance throughout MMORPG. Human players can interact with NPC just as if they are other human players.

3.4 Events

An event is an action that happens in the world and changes some state in it. Examples are a player walking, shooting a gun or casting a spell. These actions have direct and indirect consequences to the state of the world. For example a player walking changes the players own internal state which is a subset of the world state. Events contain all the information required for the game

engine to execute the event. For example, if the event is *shoot a bullet*, the included information might be the type of gun used, the location of the player shooting the bullet as well as the location of the target.

4 Goals and Challenges

We list some of the challenges faced while developing an MMORPG.

- Ability to handle a very large number of simultaneous users. As the information transferred between the players and the game server is large, the bandwidth required to support a huge number of players if enormous.
- Very large virtual worlds require huge computational power to simulate the existence of life (AI Algorithms). No single processor machine can handle the computational load required.

As it will be explained in detail later, our proposed architecture works by splitting the large virtual world into different smaller areas and assigning each area to be handled by a separate physical machine (server). Therefore both the bandwidth and computational load is spread out. Although our proposed architecture is straightforward, we faced a number of challenges specific to our architecture. These were :

- Players are not interested in receiving events that do not pertain to them. For example, a player should not receive an event update about a bullet being fired far away from the players current location.
- It is unclear what should happen when a player is near the border that separates the virtual area handled by two or more servers. Players near the border should be able to see each other and interact.
- Events that occur in an area near a border separating two or more servers can affect players that reside on different servers. Thus there exists a non-trivial probability that the game state will be invalid if an appropriate synchronization scheme is not in place.

Our goal was to develop an architecture that solves all the above problems. The hardest problem by far was to maintain consistency when events occur near the border between two or more machines.

Our design can be easily extended to be fault-tolerant. Although fault-tolerance and quick recovery from crashes is a major concern for MMORPG, it is a separate issue that escapes the scope of this paper. There exists an abundance of previous work done on fault-tolerance in real-time on-line games. Although we do not discuss these in detail, a short discussion about fault-tolerance is included.

5 Design

5.1 Overall Architecture

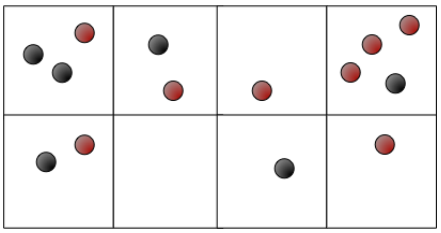

Figure 1: *The squares represent different servers each one with handling a separate area of the virtual world*

The overall system architecture builds upon the locality of interest players exhibit. Based on this players can be grouped together based on their location in the virtual world. Therefore the virtual world can be divided into smaller areas - with each area being assigned to a different

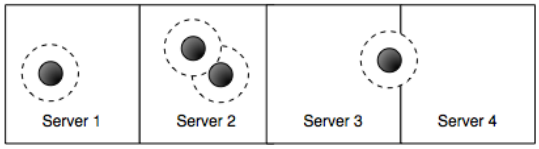

Figure 2: *The concept of an area of interest with multiple servers. From left to right, the first players area of interest lies entirely in the first server. The second and third players also lie entirely inside the second server. The fourth players area of interest spans across server 3 and server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

server as shown in figure 1. This section discusses the various design decisions

5.2 Events

We abstract all actions during a game by introducing the notion of an event. Our events follow the general principles described in section 3.4

5.3 Area of Interest

Each subdivision of the virtual world, although smaller than the entire virtual world, is still quite large. However the area of interest of a single player is very limited and usually relates simply to the sensory capabilities of the players character. Naturally, a player would not be interested in things that he can not see or hear[KLXH]. We can therefore reduce the communication required between the game server and the game client (the player) by defining *Areas of Interest*. An example is shown in figure 2.

The size of the *area of interest* depends on the player type. For example a player with a sniper rifle will have a larger area than a player without one.

Using a publish/subscribe system [FWW02, CKSW02] allows players to subscribe only for events that interest them. A novelty of our architecture is that the subscription of players to events inside their area of interest happens automatically depending on the players location inside the virtual world. The server is responsible for updating the player for any events that occur inside his area of interest.

5.4 Inter-Server Interaction

As multiple servers handle different areas of the same virtual world, there are instances when a server would be required to communicate with another server. In our architecture this is not very different from client-server communication. Servers subscribe to their neighboring servers much like players, with an area of interest of all points close enough to the border between the two servers.

For example, assume a virtual world handled by two servers, R and L, such that R handles the rightmost area and L the leftmost area. In our system, R will be subscribed to L rightmost area. The benefits of this approach will become more apparent as we discuss interaction among players near server borders.

5.5 Player on region border

A very hard yet interesting problem is when a players area of interest covers an area that spans across servers as shown in figure 2. There are four distinct scenarios

1. A player whose area of interest spans multiple servers needs to be able to see events that occur in multiple servers. For example consider a player standing near the border looking towards a region handled by a different server. The player should be able to see other players within his area of interest, even if those players are in a region handled by a different server than the players one.

2. A player while moving may suddenly move into a region that is handled by a different server

3. An event that originated in an area covered by one server may end in a region covered by a different server. For example, consider

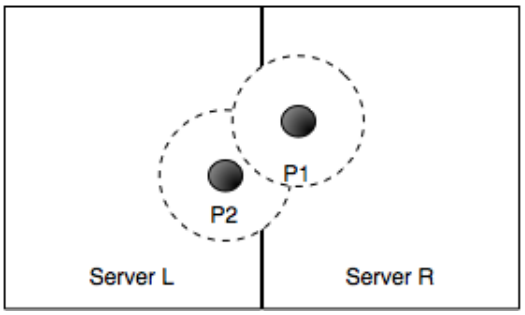

Figure 3: *An example of two adjacent game servers and two clients P_1 and P_2 each inside an area handled by a different server yet being able to see each other*

a player shooting a rocket that must land in a region handled by a different server

4. An event that occurs near the border may simultaneously affect regions covered by one or more different servers than the one it originated at. An example of this would be a large bomb exploding near the border, affecting players nearby; players could possibly reside in separate servers.

To see how our architecture handles each scenario, first recall that servers are subscribed to their neighboring servers and receive events that happen near the border. Assume a player P_1 inside an area handled by server R and a player P_2 inside an area handled by server L. Servers R and L are adjacent to each other - i.e they handle adjacent areas. This setup is shown in figure 3. We now proceed to describe each scenario.

First Case

Under this scenario if players P_1 and P_2 are near the border then they should see events originating from each other. As soon as P_2 walks to the rightmost area of server L, server R will automatically subscribe P_2 to it. P_2 will now begin to receive events in R, such as the existence of P_1.

Similarly P_1 who is inside an area handled by R is also able to receive events about player P_2 from L, reason being that P_1 is automatically subscribed to L when he enters L's area of interest.

The entire process is transparent to the player who does not notice he is receiving events from two different servers.

Second Case

Moving from one region to the other implies a transfer of game state from one server to another. Suppose P_1 moves towards the area handled by server L and eventually crosses over.

What will happen is that server R will send an event to server L and let it know that it now owns player P_1. The event will include all of P_1 state. In our architecture the players state also includes a set of all other players and servers that currently know about P_1. In our specific example this includes P_2 and servers R and L.

Events for P_1 that occur while the transfer is in progress will be forwarded to L and will arrive after the state transfer is over as the network protocol preserves the order of messages. After the state transfer has finished, L will contact P_1 with a message to contact L from now on instead of R.

The entire state transfer operation consists of a single message between the two servers. Therefore it executes very quickly and the player does not notice.

Third Case

Assume a player shoots a rocket from server L towards server R. In this case server R will receive **two** event notifications. The first event is a natural consequence of servers being subscribed to each other and is received when the rocket is shot. Server R now knows the velocity of the rocket can predict its position at any point in time. The second time server R receives the event from server L is when the rocket actually crosses the border boundary. Server R can now recalculate the position of the rocket and accept one of the two calculations - naturally it chooses

the one that allows the rocket to go through a given place sooner.

This entire process is not noticeable to the players. All players will receive event notifications about the rocket as soon as it is shot or when it enters their area of interest. Players may receive additional event notifications regarding possible effects of the rocket (i.e the rocket hit someone). In the unlikely event that an event should occur during the short time in which the state is transfered, the user will experience a delay equal to a one-way trip between the two servers. Assuming a fast network back-end connecting all the game servers, this added latency will not be noticeable at all to the end user.

Fourth Case

The fourth case although the hardest, reflects the distinctive features of the entire architecture. It is paramount to clearly conceptualize the difference between local events and shared events. Local events occur on only one server although another server may "see" them if they close to the border. Shared events affect more than one server at the same time.

The biggest challenge with shared events is to maintain consistency during the event across multiple servers. It can easily be seen that these events must affect the state of the game in a way that is equivalent to some serial order or execution of those events.

On every neighboring pair of servers, we introduce the notion of a *primary server*. Without loss of generality arbitrarily designate the *primary server* to be the one with the lowest ID. Each server takes care of ordering its own local events. The primary server is responsible for ordering among shared events. Therefore whenever a shared event originates on the primary server (such as a large bomb exploding), the primary server executes it and notifies the other interested servers. If a shared event however originates on a secondary server, it first notifies the primary server of the event and waits until the primary server sends a notification event back.

This technique guarantees serializability of the events and it introduces an additional latency to the client equal to at most the cost of a round-trip inter-server communication message in the case the event originates on a secondary server. No additional latency is introduced if the shared event originates at the primary server.

Naturally, the same technique can be generalized for border corners - locations on the world where there is an intersection of three or four different server areas. In this case we again notify the *primary server*, which has the lowest ID. Then it notifies the server with the next smaller ID and then, if needed, the latter notifies the server with the next smaller ID. Since we can arrange the servers so that at most three servers have a common border, we can obtain an upper bound on the number of inter-server messages equal to three.

5.6 Fault Tolerance

In the event of a server crash, the system should recover the entire state of the world it represents as it was prior to the crash very quickly and as transparently as possible. Our architecture is not inherently fault-intolerant, unlike P2P architectures. As every communication in the system is encapsulated inside an event, it is easy for each server to maintain a log file of all events received and in the event of a crash replay the log. Of course players that happen to be located in the area handled by the server that failed would be locked out while the server is restarting. To prevent that from happening a mirrored, replicated system as described by Cronin et al [CFKJ02] could be used.

6 Implementation

We have implemented a simple game unimaginatively called AlienBorder, to demonstrate the most important aspects of our architecture as

well as measure the relative performance gains of the distributed approach.

6.1 RPC Subsystem

AlienBorder is built on top of the Java Remote Method Invocation subsystem. Although this is inefficient from a performance perspective, it allows for a clear event-driven design that clearly demonstrates how our architecture works. JAVA RMI presents a lightweight broker to the programmer with robust facilities for remote method invocation[jav].

6.1.1 Asynchronous Calls

Java RMI does not support asynchronous remote method calls. We have therefore implemented a separate subsystem for asynchronous method invocations. The `ThreadManager` object manages a pool of threads for each server. When the server wishes to invoke a remote method without blocking, it wraps the method call inside a `Callback` object and passes it to the `ThreadManager` object, which in turn assigns it to a `HelperThread` object running in a separate thread. The method call returns immediately and the `ThreadManager` becomes now responsible for executing the call in a separate thread.

6.2 Map and Objects

The game map is simply a terrain of variable size. For our demo purposes we have implemented two kinds of objects, mobile ones that can be picked up by players (rockets, flying rockets) and static ones (planets, walls). Players are also mobile game objects.

6.3 Events and Event Handling

Our architecture requires that all updates in the state of the game occur via an event. Below is a non-exhaustive list of the events in AlienBorder :

ConnectEvent Sent from a player to a server during the initial connection

ConnectACKEvent Sent from the game server to the player confirming the connection and loading the initial player state on the game client

NewPosEvent Sent during position updates

MeetPlayerEvent Sent from the game server to the player when another player enters the first players area of interest.

TransferPlayerEvent Sent from one server to another when a player crosses the border boundary.

When an event is sent between a client and a server, vice-versa or event between two servers, it is placed in a queue. On the game client a timer which ticks as to accommodate the refresh rate of the users graphical display, executes events in the queue. On the game server events are executed immediately but they only change the internal state of the server. A timer which ticks as to simulate the passing of time then executes a procedure during which the changes in the servers internal state are reflected in the game clients and neighboring servers.

7 Evaluation

This section presents the results we obtained using *AlienBorder*, built on top of Java RMI.

7.1 Empirical Results

We run a version of *AlienBorder* on a local network of PCs using a GUI game client which drew the world on screen and allowed to user to move and fire rockets using the mouse. The GUI also displayed the server borders. As such we tried to manually create failure scenarios by simultaneously firing, constantly crossing the virtual server border back and forth, etc.

Our solution passed all tests successfully. Despite the lack of optimizations, the game was very playable.

7.2 Experimental Results

We run a version of *AlienBorder* on a large heterogeneous cluster. To do so we used Emulab [WLS+02] 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 200ms and variable packet loss rates, ranging from 0 to 0.3%.

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.

We used a game client which simulated player behavior on the virtual world. We measured the total cumulative time our program took to perform 4000 *clock ticks* in steps of 1000 ticks. From that we measured how much was spent evolving the virtual world (recall that the world evolves once per tick) and how much time was spent in handling events.

To evaluate our experimental results we used four metrics :

1. Time server required to execute 1000 virtual clock ticks[2].
2. Time the game server spent processing events it received.
3. Time the game server spent evolving the world during each tick.
4. Percentage of game events received by single client compared to the overall number of game events sent by the server over the duration of a game.

The first metric is an overall good measure of how overloaded a CPU is. A large number here shows that the server was forced to waste cycles on other tasks and could not keep up. In a real world deployment this would significantly reduce gameplay. For comparison we show the results of a single server with no clients. The second metric shows if the game server is forced to spent most of the time processing events received by clients. The third metric is similar to the previous one but shows if the game server is forced to spent too many CPU cycles at each tick. Lastly, the fourth metric shows if *areas of interest* have helped reduced the number of events received by the client and thus are a useful measure of not only CPU consumption by the clients but also of the network bandwidth usage per player.

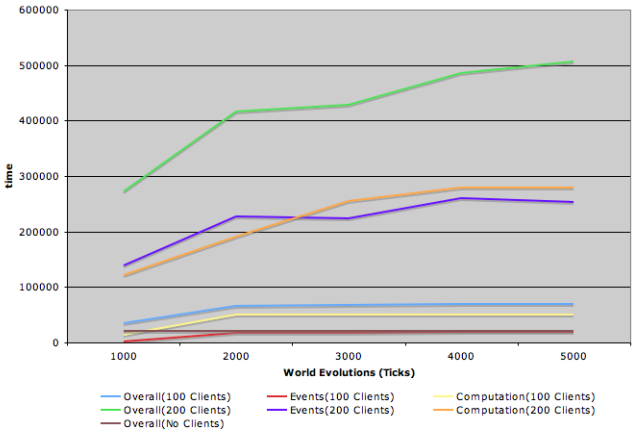

Figure 4: *The performance of a single server when one hundred players and when two hundred. All server metrics are displayed. It can be seen that the server performs much worse as more clients gradually connect. In the end it requires six times more than optimal to evolve the world at each tick*

Figure 4 shows the results when testing a single server. It can be easily seen that a single server can not scale very well. Figure 5 shows the results when testing our architecture with just three servers and four hundred clients. Our architecture with three servers performs vastly better than a single server architecture, being able to handle twice as many players and taking a smaller performance hit at the same time.

[2]evolve the world 1000 times

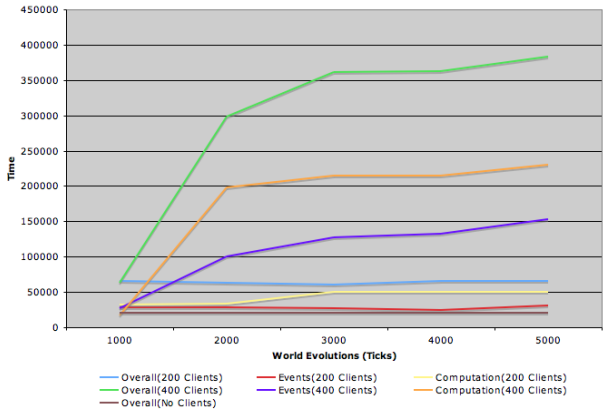

Figure 5: *The performance of three servers using our architecture when two and four hundred players. With four hundred clients our architecture needs just four times more than optimal to evolve the world at each tick*

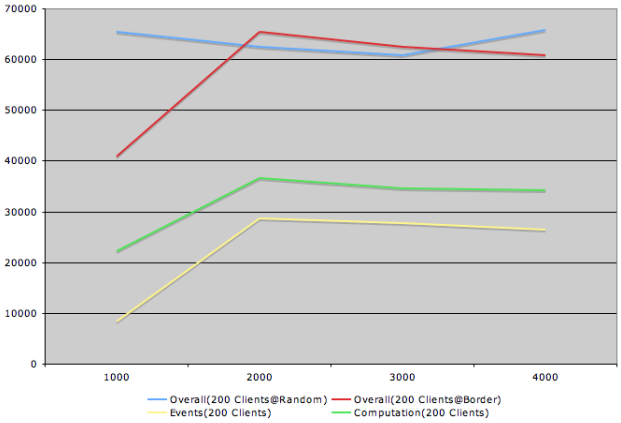

Figure 6: *The performance of three servers using our architecture with two hundred players. This time the players initialized and remained near the virtual server borders. For comparison the previous results from the normal case when players spawned randomly are included.*

Figure 6 compares our architecture when a large number of players are located near the border. Our solution for the border case performs very well, no worse than when players are dispersed randomly in the game world.

Evidently, our results have shown that our distributed architecture performs significantly better than a non-distributed one. In addition to that, we show that even with heavy activity near server borders our architecture does not take a significant performance hit.

8 Conclusions and Future Work

In this paper we present a novel centralized, distributed architecture for MMORPG. 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 a solution to the problem of handling game events occurring near virtual borders is shown and a proof-of-concept demo is provided.

Even though our architecture was built with MMORPG in mind, there is no reason why it can not be extended to other types of games such as First Person Shooters.

In conclusion, we have shown that a new game written with our architecture in mind can perform much better than a game written as a simple client/server and discuss why non-centralized peer-to-peer have problems that make them non-pragmatic for real-world commercial deployment.

In the future, much work is needed especially in the area of fault-tolerance. We are experimenting with techniques such as log files and replication to see how well they perform inside our design. In addition, this paper does not account for persistent world states stored in databases or very computationally intensive game AI algorithms. Our architecture however could be extended to support all the above.

References

[BRS02] Ashwin R. Bharambe, Sanjay Rao, and Srinivasan 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.

[CFKJ01] Eric Cronin, Burton Filstrup, Anthony R. Kurc, and Sugih Jamin. A distributed multiplayer game server system. May 2001.

[CFKJ02] Eric Cronin, Burton Filstrup, Anthony R. Kurc, and Sugih 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.

[CKSW02] Sergio Caltagirone, Matthew Keys, Bryan Schlief, and Mary Jane Willshire. Architecture for a massively multiplayer online role playing game engine. *J. Comput. Small Coll.*, 18(2):105–116, 2002.

[FWW02] Stefan Fiedler, Michael Wallner, and Michael 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.

[HB03] Tristan Henderson and Saleem Bhatti. Networked games: a qos-sensitive application for qos-insensitive users? In *RIPQoS '03: Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS*, pages 141–147, New York, NY, USA, 2003. ACM Press.

[jav] *Java Distributed Systems Home Page*. http://www.sun.com/rmi?

[KLXH] Björn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games.

[LP04] Hunjoo Lee and Taejoon Park. *Lecture Notes in Computer Science*, chapter Design and Implementation of an Online 3D Game Engine, pages 837–842. Springer-Verlag GmbH, 2004.

[Mac04] Dean Macri. The scalability problem. *Queue*, 1(10):66–73, 2004.

[SSR+04] Anees Shaikh, Sambit Sahu, Marcel Rosu, Michael Shea, and Debanjan Saha. Implementation of a service platform for online games. In *SIGCOMM 2004 Workshops: Proceedings of ACM SIGCOMM 2004 workshops on NetGames '04*, pages 106–110, New York, NY, USA, 2004. ACM Press.

[WLS+02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet 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, December 2002. USENIX Association.