Optimistic Load Balancing in a Distributed Virtual Environment

Roman Chertov and Sonia Fahmy[*]

ABSTRACT

Distributed virtual environments such as massive multi-player games require multiple servers to balance computational load. This paper investigates the architecture of a unified environment where the virtual online world is not partitioned according to rigid boundaries, but according to an adaptive paradigm. Since it is difficult to develop an optimal load balancing algorithm for a unified environment, we propose an optimistic scheme that quickly converges. The cost of frequent migrations is reduced by following a push/push data exchange model. We analyze the computational time costs of such a system and give simulation results to gauge its performance. The simulation results confirm that our load balancing scheme is efficient and can support large numbers of clients.

1. INTRODUCTION

Many distributed virtual environments (DVEs) for collaborative research, interaction, and entertainment are being deployed over the Internet. These online interaction and gaming DVEs are becoming more popular, creating a multi-billion dollar industry. A straightforward approach to set up a large scale DVE is to partition the virtual space into fixed areas, where clients (e.g., game players) in one area cannot see what is occurring in the adjacent area until they reach that area. This poses the architectural question of how to support thousands of clients connected to a single *unified* virtual environment that is *not rigidly partitioned*.

To address this question, we consider the DVE as a distributed database where clients are remote sites with the server being a concurrency and replication controller. A fundamental problem in distributed databases is the quality of the communication channel. In environments requiring hand-eye coordination, latency is a key factor as it determines how many updates per second the DVE can perform. Ideally, the DVE should sustain 30 updates per second or higher, as otherwise clients will notice uneven motion in their own as well as other client movements [4]. Although it is possible to achieve acceptable performance with lower update rates by relying on direction vectors and prediction [1], such schemes fail when the client rapidly shifts direction.

A minimum Quality of Service (QoS) level is needed for high DVE performance. Since integrated and differentiated services are not widely deployed in the current Internet, our architecture employs application level routing/QoS. Each Internet service provider (ISP) can provide one or more *gateway* nodes [2] to which its customers connect to participate in a DVE (similar to Akamai nodes for WWW caching). Figure 1 depicts two gateways at two different ISPs which connect to the same *server core* (or server cluster). The gateways can act as client data aggregation points. In addition to providing fast access to the server, gateways can be pruning/filtering points. Further, the gateways can act as synchronizers to ensure that all clients receive updates at approximately the same time to maintain fairness.

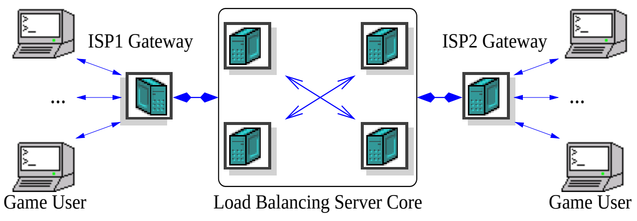


Figure 1: System architecture

Creating and managing such gateways is only worthwhile if the server core can support an extremely large number of clients. Customers will be reluctant to pay subscription fees if most of the time the server is too busy and they cannot participate. Therefore, the server core must be a cluster of machines that perform *intelligent load balancing* to evenly share the client load in a scalable manner. We employ a push/push data exchange model to reduce the cost of client migration among servers.

This paper gives an architecture and algorithms for a DVE similar to the system depicted in Figure 1. The system allows the clients to move with selected directions and speeds, with the assumption that travel along the up/down axis is not significant, meaning that the world can be examined from the top down as a 2D map. This is well-suited for the majority of interactive systems on the market today. We design and evaluate the performance of a novel adaptive load balancing scheme for the server core that exploits the fact that clients *tend to cluster around points of interest*.

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 gives an overview of the require-

[*]Roman Chertov and Sonia Fahmy are with the Department of Computer Science, Purdue University, 250 N. University St., West Lafayette, IN 47907–2066, USA. Tel: +1-765-494-6183. Fax: +1-765-494-0739, E-mail: {rchertov,fahmy}@cs.purdue.edu. This work has been sponsored in part by NSF grant CNS-0238294 (CAREER).

ments and architecture of the system. Section 4 analyzes the computational cost. Section 5 explains our experimental setup. Section 6 discusses our preliminary results. Finally, Section 7 summarizes our conclusions and planned future work.

2. RELATED WORK

Large scale virtual environments cannot be effectively managed on a single server, even with advent of multi-core CPUs. This is because a large scale environment can include thousands of clients and the processing power of a single powerful machine can still be inadequate. To address this scalability requirement, three approaches are dominant: (i) peer-to-peer [12, 5, 13], (ii) server cluster [8, 16, 11, 2, 7, 6], and (iii) distributed servers [5, 15, 17]. Out of the three approaches, the server cluster may be the most expensive to implement, and can be a point of failure. However, the cluster approach can offer much better latency guarantees when coupled with application specific routing, traffic engineering, and gateway nodes [2]. From the business point of view, such a setup allows easier accounting and can deliver guaranteed performance to clients.

A straightforward approach to divide load among servers in a cluster is to use a grid where each server manages a collection of cells [17, 11, 8, 7]. The servers migrate cells to distribute the load. The cell size is configurable and can be set to be equal to the field of view (FOV) of a client, as suggested in [8]. However, if the DVE is very large and has large regions of empty space, then the load balancing scheme can be extremely sub-optimal. An alternative approach to the problem is to subdivide the virtual space into evenly sized cells and then transfer cell contents to arbitrary servers to balance the load [19]. Our approach is more adaptive, as cells need not be evenly sized.

Peer-to-Peer approaches such as [15, 12, 13] rely on clients to compute the adjacency lists, thus removing the need for a server to perform such computations. However, these approaches cannot satisfy latency requirements and can suffer if client densities are high and client movements are fast. Use of TCP connections as discussed in [12] is prohibitive in highly dynamic environments, as frequent connection setup and tear-down will significantly degrade performance.

Lossless data delivery is not a strict requirement of many DVEs. Presence of some loss has been shown to be acceptable by [4, 5, 9]. As long as loss is not too frequent and important events are only rarely lost, clients are likely to find the DVE performance more than adequate. Analysis of a first person shooter game, Counterstrike, revealed that the worst tolerable loss is between 1-2% and the game generates highly periodic traffic [9].

3. SYSTEM OVERVIEW

DVE clients do not typically move according to a random walk [18]: there is always some structure to the overall position of clients. Our system exploits the fact that clients tend to cluster around points of interest which vary from one virtual environment to another. The points of interest can be *dynamic* and unknown ahead of time, making it hard if not impossible to partition the game space ahead of time, e.g., [7]. Hence, a highly dynamic load balancer is required. A good load balancer must satisfy the following properties: *(i) achieve even load distribution, (ii) efficiently handle sparse environments, and (iii) allow dynamic points of interest.*

Figure 2 illustrates a DVE with 8 clients. The dotted circles around the clients represent the field of vision (FOV) range. The arrows represent the client direction vectors. Given three servers, it is possible to construct rectangular regions that divide the client processing evenly among the servers. We have chosen rectangles

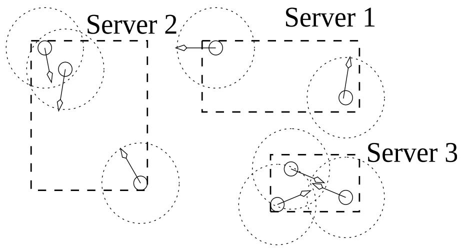


Figure 2: Ideal client partitioning

as it is fast to perform geometric operations on them, compared to more complex polygons. The rectangular coverage regions can change in size as the clients travel, but as long as clients stay in the proximity of the point of interest, there is no need to change server-client assignments. Until the coverages of servers intersect or are in a client FOV proximity of each other, the servers do not need to interact extensively to determine if the clients in their domain can see other clients.

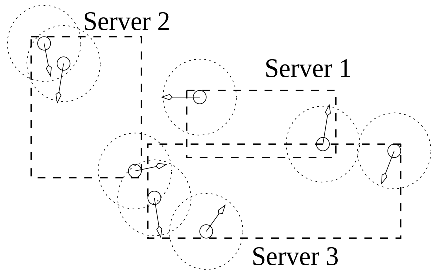


Figure 3: Coverage overlap due to using a rectangle as a bound

Clearly, perfect partitioning cannot always be maintained. Clients typically remain near a point of interest for some time, but then choose to move to another point, causing overlap. Figure 3 depicts a scenario where a client at Server2 can see a client at Server3 and vice versa. This can be easily checked by keeping track of which client FOVs go outside the coverage area. The case when Server1 and Server3 overlap is more problematic. Searches have to be conducted inside the coverage areas, and this can become expensive if the number of clients is large or if overlaps are frequent. We discuss this problem further later in this section.

In the remainder of this section, we describe the complete architecture of the load balancing algorithm, server core, and gateways.

3.1 Load Balancer

Individual servers in the server core perform greedy local actions to resolve overlap problems such as those shown in Figure 3. The optimism in the system is derived from the fact that we quickly reach a good global state which we predict will remain stable for several cycles. The load balancer has two modes of operation. In the first mode, the balancer attempts to balance the load. The second mode is needed to resolve any overlap inefficiencies.

Each server uses a *client_threshold* value to determine the number of clients it is willing to serve. If *client_threshold* is exceeded, the server attempts to migrate part of the load to a nearby server. In certain situations, a server can accept more clients than specified by *client_threshold*, but it will not accept any more when *max_clients* is reached. Figure 4 demonstrates migration in action. The number of clients on Server2 exceeds the *client_threshold* value of 3 and Server2 starts migration. The server selects a server that is closest

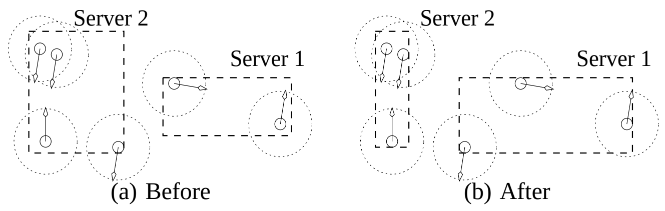


Figure 4: Client migration when number of clients exceeds the *client_threshold* of 3

and has not reached its maximum capacity. Then the client that is closest to that server coverage area is transferred. We have placed a condition on the transfer to avoid excessive overlap: the transfer will not occur if the new area of the server will exceed *max_area*, which is configurable. This keeps the system in check, so that a server cannot begin a rapid massively overlapping expansion.

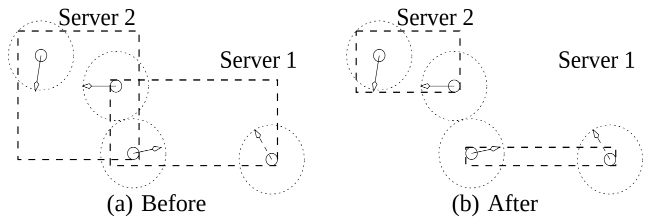


Figure 5: Coverage intersection resolution

Migrations and client movements will inevitably produce cases where coverage areas overlap. Figure 5 illustrates a scenario when a client from Server1 moves into the coverage area of Server2. To resolve this situation, each server examines servers that overlap with it and attempts to transfer all of its own clients that are in the overlapping region to the other server, provided that maximum capacity has not been reached. One heuristic that we apply in this case is a random scan of the intersecting servers. This is important in situations when the same region is in the coverage of more than two servers. Random scan ensures that various combinations of intersection resolution are carried out until a good solution is found.

To further reduce coverage overlap, we introduce one additional rule. If the area of coverage of the server exceeds *max_area*, then *client_threshold* is set to 0 so that migrations will occur. This is necessary to avoid cases when the coverage area of Server X begins to grow causing increasing overlap, and servers that were not previously neighbors of X start performing transfers to it, further compounding the problem.

The algorithm parameters thus allow a *tradeoff* between *load variance reduction* and *overlaps*. If the value of *max_clients* is close to *client_threshold*, load variance among servers will be low; however, this will leave little room for coverage intersection resolution, resulting in a larger number of overlaps.

3.2 Server Core

The server core serves as an aggregation point of all client data, and employs the load balancing algorithm described above. In addition, the server core can process client data by running a limited version of the DVE engine with all the graphics routines stripped. By running this engine on the server core, cheaters [3] can be detected since client and server calculations would deviate in that case. In an alternate setup, the server core can run the DVE engine and the clients only process input and display functions. This is useful in situations where clients have limited CPU capacity, e.g., clients are handheld devices.

To reduce the cost of client migration within the server core, we employ a *stateless* mode, meaning that individual servers do not store client information locally. This eliminates the need for complex data migration architectures as in [7]. It also eliminates migration delay. Our stateless approach is suited to cases when a significant portion of the client state will change with each update, and the state can be compactly represented (since the DVE type/rules are known). The server maintains *a pool of client slots* to fill with information as updates and inserts occur. The slot pool eliminates memory fragmentation problems. Observe, however, that if the update to player state ratio is low, the stateless mode may be undesirable since traffic volume will be too high. This can be mitigated by using different levels of detail such that full state is not required until the last possible moment.

The server core communicates with *gateways* that perform application level routing. Stateful connections are avoided: UDP packets are routed by simply looking up the server unique identifier (ID). Efficient application level routing can be accomplished by a modular software router like Click [14] or a network processor. Hardekopf et. al. [10] report that increasing number of microengines in network processors will allow handling application specific classification and routing at line speeds.

The gateways first *push* client update data to the servers. Each client has a gateway ID and a server ID associated with it. If a client migration occurs, the server core will *push* new data towards the gateways reflecting the new server to client relationship. This forms the basis of our stateless *push/push* data exchange model.

3.3 Gateways

Each gateway acts as a high speed connection point to the server core from a particular ISP. A gateway can be a fast server or a collection of network processors [2] that are optimized to handle the specifics of the DVE. The gateway synchronizes with the server core, ensuring that clients use the same global time. This is necessary to compensate for latency effects when using prediction and direction vectors [1]. The gateway can also delay packets destined to clients to ensure that all clients receive packets at the same time. This prevents cases when some clients can see events before the other clients can. Although the clients can adhere to equal cycle times, the gateway should not be forced to wait for all the client data to arrive before it pushes it to the server core, in order to avoid slow down due to ill performing clients. This can result in incomplete updates, but as long as clients do not miss their scheduled sending times too often, this problem will not be noticeable [9, 5].

Each gateway receives data from the server core only regarding the clients that belong to it and what they can see. This filters out significant irrelevant information, reducing processing and bandwidth costs. The gateway must also filter the data that is relevant to each client, as it would be inefficient to forward the entire dataset to all its clients. Filtering can be performed based on the client FOV. If the client has no possible way of seeing an object, then there is no need to pass that information along.

4. COMPUTATION TIME ANALYSIS

We first consider the case of a single server, then a naive approach with multiple servers, and then our optimized approach.

4.1 Single Server

In the case of a single server that uses the gateway model described earlier, the server must compute the DVE physics for each client, and then compute which clients can see each other so that it can prune the information the gateways receive. Let N be the number of clients. Let P_i be the time required to process physics and cheat detection for $Client_i$. Let $C_{i,j}$ be the time required to compute if $Client_i$ and $Client_j$ can see each other and if they have collided. In a naive single server system performing pair-wise comparisons, the total time, T_s, can be computed as $T_s = \sum_{i=0}^{N} P_i + \sum_{i=0}^{N} \sum_{j=0}^{N} C_{i,j}$. If we assume that $P_i = P$ for all i, and since each $C_{i,j}$ operation is constant time, we get $T_s = NP + N^2$.

4.2 Multiple Servers

Consider a system with M servers and N clients. Assume that N_i is the number of clients for $Server_i$, and (as above) that P is the time required to process physics and cheat detection for any client. The cost of tasks to be performed by server i, T_i, includes:

1. $N_i P$ to process the physics and cheat detection for its own clients.

2. N_i^2 to search for local client interactions.

3. $M - 1$ to determine the K intersecting and adjacent rectangles (within client FOV) that are managed by other servers.

4. $N_i \sum_{k=0}^{K} N_k$ to determine the clients that potentially can interact with its clients.

5. $N_i + K$ to determine the client that is closest to the closest server if migration is required.

6. KN_i to determine which current clients are within other servers' coverage so that they can be migrated there, provided the maximum capacity has not been reached.

The total time of the system is computed as $T_{total} = max(T_0, \cdots, T_M)$.

4.3 Optimized Cost

The simple analysis above shows that if K is large or client load distribution (N_i is large for some i) is uneven, the multi-server system will perform worse than the single server system. K is highly dependent on the number of overlapping rectangles, because if there was no overlap and all servers covered disjoint spaces, there would be at most 8 neighbor rectangles to check. Hence, K depends on the distribution of clients and the performance of the load balancer.

The cost of client proximity searches can be reduced to a worst case of $O(N2N^{0.5})$ if a point quad tree is used. It also takes $O(NlnN)$ to build the tree if approximately random insertion is assumed. Therefore, it would be faster to build a tree from scratch and perform proximity searches than doing pair-wise comparisons. Modifying the steps in Section 4.2, we now obtain the following equation for the time of each server: $T_i = N_i P + 2N_i^{1.5} + N_i ln N_i + M - 1 + (2N_i \sum_{k=0}^{K} N_k) \times (N_i \sum_{k=0}^{K} N_k)^{0.5} + (N_i \sum_{k=0}^{K} N_k) ln(N_i \sum_{k=0}^{K} N_k) + N_i + K + KN_i$. The computational complexity is thus: $O(N_i^{1.5} + (N_i \sum_{k=0}^{K} N_k)^{1.5})$ if we assume constant time P. When using the same optimization (and again assuming constant time P), the complexity of a single server system is $O(N^{1.5})$. Hence, it is clear that it is *essential* to minimize K, and keep N_i as close to N/M as possible. For a large value of N, the main cost is governed by the client interaction algorithm, and *not* the physics processing.

5. EXPERIMENTAL SETUP

We have implemented a simulator of the entire system [1] to evaluate the load balancer performance. The current version of the simulator consists of only a single gateway to which all clients connect. The gateway is connected to the server core as in Figure 1. The core consists of a number of servers that perform the load balancing operations described in Section 3.1. The system was fixed to run at 30 updates per second, meaning that data exchange and load balancing occurs thirty times a second. A real system may not be able to run as fast. If the average latency to the server core is 100 ms for example, then only 10 updates per second is feasible.

We increase the number of clients in increments of 100 from 100 to 2000. Each experiment was run 10 times with the same settings but different seeds and the results were averaged. To gauge the effectiveness of the system, we measure the total area covered, server loads, number of overlaps, and overlap area.

5.1 DVE Environment

We use a ten by ten kilometer map that has 64 points of interest evenly distributed on the map in a grid-like fashion. We use a core composed of 64 servers. The values of *client_threshold*, *max_clients*, and *max_area* were computed as *number_of_clients*$/64$, $1.5\times$ *client_threshold*, and $10\ Km \times 10\ Km/60$ respectively. This is akin to a large "Battlefield 1942" map with 64 control points. Additionally, the servers were allowed a maximum of 3 migrations per cycle. The experiment duration was 30 minutes to simulate DVEs where map rotation occurs frequently.

5.2 Client Behavior

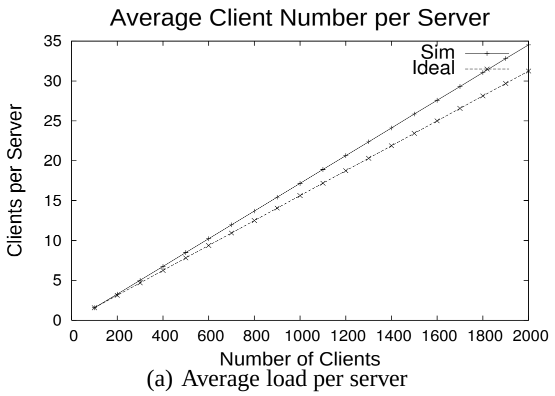
We considered two models for client mobility. In the first model, the clients are uniformly and randomly spawned. Each client performs a random walk with a random speed, direction, and distance. The client then travels along the chosen path and when the selected distance is traversed, the client chooses a new set of parameters and continues. This leads to scenarios where the density of clients throughout the entire region is close to uniform. Since such behavior is unrealistic for DVE clients, we have devised a second more sophisticated model which we used in all experiments reported below. Just as in the previous case, the clients are randomly spawned. However, the map now includes points of interest to which clients will flock. A client randomly chooses a point of interest in a $\frac{map\_length}{3}$ radius and goes there. While en-route, it can change its mind with a small probability of 0.001 and select another destination. Clients are allowed to move at a maximum speed of 5 m/s which corresponds to human jogging speed. Upon arriving at the destination, the client performs a random walk around the point of interest (i.e., stays within a 50 m $\times$ 50 m region). When it has walked for 300 meters, it selects a new destination and repeats the process. Such waypoint-like behavior is more realistic as clients tend to gather near points of interest (resources, battles, etc.), and then go elsewhere. Since in our model (i) clients prefer waypoints that are closer, (ii) clients can change direction while en-route, and (iii) the map has no geographical barriers, our model addresses the shortcomings of the basic "Random Waypoint Model" as discussed in [18].

6. PRELIMINARY RESULTS

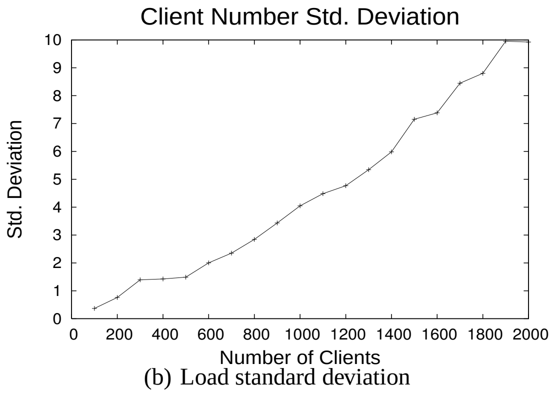
The prime measure of a load balancer is how evenly it distributes the load. Thus, we measure the average load per server throughout the experiment duration and then average the final results. We also

[1]The simulator can be freely downloaded from http://www.cs.purdue.edu/homes/rchertov/.

compute the standard deviation to gauge variability. Figure 6(a) shows how close the results are to the ideal (precisely even) balancer. We also observe that the standard deviation steadily increases with larger number of clients, meaning that the load balancer allows for more load variance as density increases. The variance is highly dependent on the *max_clients* parameter. If *max_clients* is too low, then variance is reduced but the number of overlaps increases, as it becomes more difficult to perform overlap resolution. This is the tradeoff discussed in Section 3.1. We selected *max_clients* to be $1.5\times$ *client_threshold*, accepting an increase in variance in return for fewer overlaps. However, even with the current settings, the system approximately satisfies property (i) in Section 3 (achieve even load distribution).



(a) Average load per server

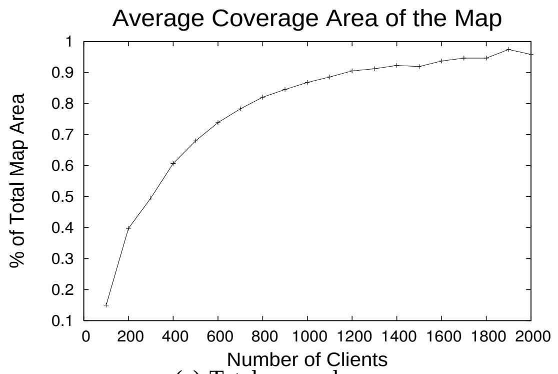


(b) Load standard deviation

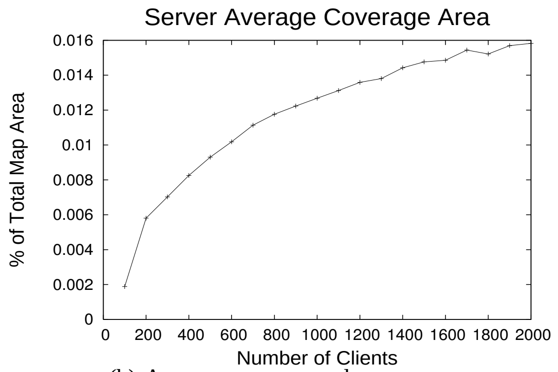
Figure 6: Server load results

In addition to achieving good load balancing properties, our system must efficiently handle sparse areas, avoiding the drawbacks of grid-based systems. To compute the areas of overlap, we find all intersecting pairs and then sum the overlapping areas. The resulting sum can be larger than the geometric overlap area when more than two servers overlap. This allows estimating the performance of the intersection resolution algorithm more accurately. Figure 7 shows that our balancer performs well. As the client density in the environment increases, so does the total area covered by all of the servers. This means that if the size of the environment is very large, then empty regions can be ignored. This satisfies property (ii) (efficiently handle sparse environments).

To validate that property (iii) (allow dynamic points of interest) is achieved, we conduct experiments with 2000 clients where the points of interest are randomly chosen on the map. We studied 10 different runs with the same parameters but different seeds and found the average standard deviation to be 8.6 and average server



(a) Total covered area



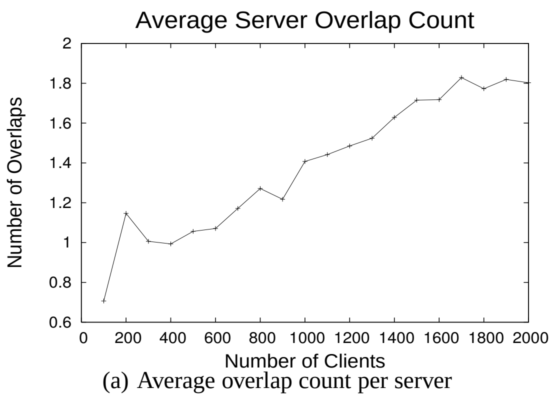
(b) Average area covered per server

Figure 7: Area coverage results

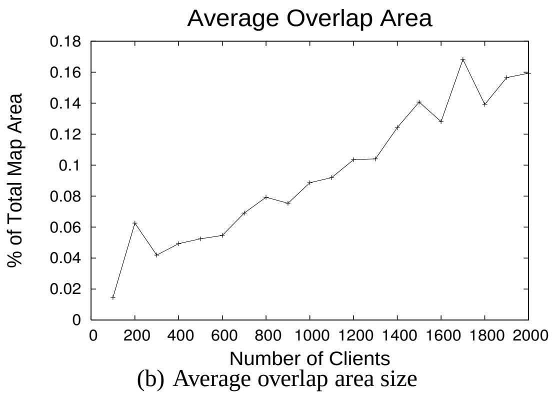
overlap count (number of servers with overlapping coverage) to be 1.83, with small variance between the runs. This fact highlights the ability of the system to deal with dynamic clustering of clients, as there is no requirement for clusters to be uniformly spaced or known ahead of time.

Another key property of a good dynamic load balancer is the ability to minimize intersection of server coverage areas, as overlap entails that servers query other servers regarding their clients. Figure 8 shows that the average overlap count for each individual server only increases by 2.5 from 100 to 2000 clients; however, the average area of the overlapping regions increases by a factor of 8. The reason for this is that when only 100 clients are on the map, there are 64 servers which can very efficiently partition the space. As the number of clients per server increases, so does the coverage area, leading to an increasing area of overlaps, as discussed above. It is important to note that the server overlap count in this section is the same as the variable K in Section 4.2. The figure shows that it is slowly increasing with the number of clients, and good load distribution is achieved. This supports our conclusion in Section 4.3 that a multi-server approach will outperform a single server system as long as K is small and server load is close to even.

Observe that the clients in the above experiments moved at 5 m/s which corresponds to human jogging speed. We have conducted several experiments when the maximum speed was set at 50 m/s (or 180 Km/h), corresponding to vehicle speed. As expected, overlap increased, but by only a small margin. In another test, we reduced the number of servers to 25 but kept the points of interest at 64. As expected, this led to an increase in overlap area but the average overlap count per server was slightly lower than in the fast client experiment (1.95 versus 2.20).



(a) Average overlap count per server



(b) Average overlap area size

Figure 8: Overlap results

Finally, we conducted experiments with 10000 clients, 100 servers, and 100 points of interest. We obtained a load standard deviation of 35.2 and an average overlap count of 2.67, proving that our system has the potential to scale to a very large number of clients.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have investigated the architecture of a unified environment where the virtual online world is not partitioned according to rigid boundaries, but according to an adaptive paradigm. Our analytical and simulation results indicate that our load balancer satisfies the three load balancing properties outlined in Section 3: (i) achieves even load distribution, (ii) efficiently handles sparse environments, and (iii) allows dynamic points of interest.

Our system can be extended in several ways. Representation of the core as a single server at a top level can lead to good hierarchical scaling properties, allowing the system to support thousands of clients seamlessly in a joint virtual world. A more flexible load balancer can dynamically allocate/deallocate servers depending on the load. Adjusting the *client_threshold* and *max_clients* values of individual servers can balance the load distribution tradeoff, and take into account different underlying sever machine speeds.

8. REFERENCES

[1] S. Aggarwal, H. Banavar, A. Khandelwal, S. Mukherjee, and S. Rangarajan. Accuracy in dead-reckoning based distributed multi-player games. In *NetGames '04*, pages 161–165, 2004.

[2] D. Bauer, S. Rooney, and P. Scotton. Network infrastructure for massively distributed games. In *NetGames '02*, pages 36–43, 2002.

[3] N. Baughman and B. Levine. Cheat-proof playout for centralized and distributed online games. In *INFOCOM 2001*, volume 1, pages 104–113, 2001.

[4] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool. The effects of loss and latency on user performance in unreal tournament 2003. In *NetGames '04*, pages 144–151, 2004.

[5] A. Bharambe, J. Pang, and S. Seshan. A distributed architecture for interactive multiplayer games. Technical Report CMU-CS-05-112, Department of Computer Science, Carnegie Mellon University, January 2005.

[6] David Brandt. Networking and scalability in EVE online. NetGames '05 Keynote, 2005.

[7] Glen Deen. Making quake II massively multiplayer with optimalgrid. ICPP '05 Keynote, 2005.

[8] T. Duong and S. Zhou. A dynamic load sharing algorithm for massively multiplayer online games. In *ICON 2003*, pages 131–136, October 2003.

[9] W. Feng, F. Chang, W. Feng, and J. Walpole. Provisioning on-line games: A traffic analysis of a busy counter-strike server. In *IMW '02*, pages 151–156, 2002.

[10] B. Hardekopf, T. Riche, J. Kaur, J. Mudigonda, M. Dahlin, and H. Vin. Impact of network protocols on programmable router architectures. Technical report, University of Texas, Austin, November 2002.

[11] M. Hori, T. Iseri, K. Fujikawa, S. Shimojo, and H. Miyahara. Scalability issues of dynamic space management for multiple-server networked virtual environments. *PACRIM 2001*, 1:200–203, August 2001.

[12] S. Hu and G. Liao. Scalable peer-to-peer networked virtual environment. In *NetGames '04*, pages 129–133, 2004.

[13] 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*, pages 116–120, 2004.

[14] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[15] J. Lui and M. F. Chan. An efficient partitioning algorithm for distributed virtual environment systems. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):193–211, 2002.

[16] D. Min, D. Lee, B. Park, and E. Choi. A load balancing algorithm for a distributed multimedia game server architecture. In *ICMCS'99*, volume 2, page 882, 1999.

[17] B. Ng, R. Lau, A. Si, and F. Li. Multiserver support for large-scale distributed virtual environments. *IEEE Transactions on Multimedia*, 7(6):1054–1065, December 2005.

[18] S. Tan, W. Lau, and A. Loh. Networked game mobility model for first-person-shooter games. In *NetGames '05*, pages 1–9, 2005.

[19] B. Vleeschauwer, B. Bossche, T. Verdickt, F. Turck, B. Dhoedt, and P. Demeester. Dynamic microcell assignment for massively multiplayer online gaming. In *NetGames '05*, pages 1–7, 2005.