RC23764 (W0510-231) October 28, 2005
Computer Science

IBM Research Report

Matrix: Adaptive Middleware for Distributed Multiplayer Games

Rajesh Krishna Balan
Carnegie Mellon University

Archan Misra, Maria Ebling, Paul Castro
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Matrix: Adaptive Middleware for Distributed Multiplayer Games

Rajesh Krishna Balan[†], Archan Misra[‡], Maria Ebling[‡], and Paul Castro[‡]
[†]Carnegie Mellon University and [‡]IBM Research Watson
rajesh@cs.cmu.edu, {archan, ebling, castrop}@us.ibm.com

Abstract

Building a distributed middleware infrastructure that provides the low latency required for massively multiplayer games while still maintaining consistency is non-trivial. Previous attempts have used static partitioning or client-based peer-to-peer techniques that do not scale well to a large number of players, perform poorly under dynamic workloads or hotspots, and impose significant programming burdens on game developers. We show that it is possible to build a scalable distributed system. This system, called Matrix, is easily usable by game developers. We show experimentally that Matrix provides good performance, especially when hotspots occur, and that even with simple algorithms, Matrix's performance is acceptable to game players.

1 Introduction

Online gaming is a rapidly growing market segment (estimated to be USD $5 billion with 100 million players by 2008 [9]), with major companies such as Microsoft (*Xbox Live*) and Sony (*PS2 Online*) currently devoting significant resources towards online multiplayer gaming infrastructures. A particularly interesting form of multiplayer gaming is the rapidly growing [29] class of massively multiplayer online games (MMOG) such as *Everquest* [22] and *Final Fantasy XI* [23], where hundreds or even thousands of players from all across the world interact in real-time in a shared virtual world.

To support these virtual worlds, most MMOGs currently use a centralized server model, with players connecting to a single game server that handles the entire game world. However, studies show that an individual server can handle at most 30,000 clients [7] whereas games like Final Fantasy XI claim to have at least one million registered players [24]. To handle more players, some MMOGs [7] use multiple servers that are statically assigned different parts of the game world, even though this approach is known to be unresponsive to unexpected workload variations or dynamic localized hotspots in the game.

To overcome this limitation, static partitioning schemes either significantly overprovision the number of servers used for the game and/or impose artificial limits on the number of players that can be in any part of the map. Unfortunately, overprovisioning incurs extra costs and artificial limits may detract from the gaming experience. It would be better to instead, use a distributed system that can handle arbitrary game loads by dynamically and automatically adjusting the number of servers used by the game in a scalable and efficient manner. This system could either be used on its own or in combination with static partitioning schemes (as a mechanism to handle unexpected load changes).

Building this dynamic distributed system for MMOGs, however, is a non-trivial problem. To preserve the interactive feel of a MMOG, the client response latency must be low [3]. But, maintaining complete consistency between distributed nodes imposes quadratically increasing amounts of time as the amount of traffic and number of nodes in the system increases (due to increased player activity). On the other hand, a lack of consistency could lead to an unsatisfactory experience for the game player. The challenge lies in satisfying these conflicting latency and consistency goals, especially for a system with a large number of nodes and a high volume (O(Gbps)) of network traffic.

The key insight that allows us to overcome this problem is the observation that MMOGs are an example of a *nearly decomposable system* [21]. Such a system is one in which the number of interactions among subsystems, in some geometric space, is of a lower order of magnitude than the number of interactions within an individual subsystem. For MMOGs, this behavior typically manifests itself through a "radius" or "zone of visibility" associated with each game player. It is usually sufficient to update players with only those events that occur in their zone of visibility. For example, if a tank is destroyed in a battlefield game, it is enough to only send this information to other tanks that can see the victim, rather than to all the tanks in the game.

Using this insight, we built a scalable low-latency distributed middleware infrastructure, called *Matrix*, that provides pockets of locally-consistent state. This weaker form of consistency allows Matrix to provide low latency responses, while still giving adequate consistency to game clients even when the number of nodes in the system increases. Matrix also provides low latency mechanisms to handle infrequent global interactions. Ease of use is another key design goal of Matrix. We achieved this by providing a clean and clear layering that hides the consistency maintenance details within an easy-to-use API. This API allows Matrix to be used with only minimal changes to existing MMOGs. The layering also allows Matrix to support the distributed operation of various MMOGs *without actually needing to understand the game logic*. Finally, unlike static partitioning techniques, Matrix can dynamically add and remove servers as necessary to handle transient hot-spots and dynamic loads caused by players joining and leaving the game.

We validated both Matrix's system-level performance as well as its effectiveness at satisfying real game players. In particular, we show that Matrix's overhead is reasonable and also that it outperforms a statically partitioned system when unexpected load patterns occur. Finally, we show, via a small user study, that Matrix, even though it intentionally uses simple algorithms, is still able to satisfy real game players.

In Section 2, we describe Matrix's design criteria while Section 3 presents the design and implementation of Matrix. Section 4 describes the Matrix API. Section 5 presents evaluation results while Section 6 presents related work.

2 Matrix Design Criteria

In this section, we describe the two key design criteria (and their corresponding implications) used to build the Matrix middleware. In particular, Matrix was specifically designed to allow MMOG game developers to focus mainly on their game's core logic and delegate the task of scalably distributing the game to Matrix.

2.1 Attractive and Easy for Game Developers

The first key criteria was to make Matrix attractive for game developers to use. Most game companies usually focus on core game-specific technologies, such as 3D graphics modeling, and typically have very little in-house distributed systems expertise. Hence, being able to leverage a distributed game middleware that scales and maintains adequate consistency as the user population grows would be of great benefit for them. To appeal to developers, Matrix has the following characteristics:

No Change in Security Model : A primary concern for online game developers is cheating and denial-of-service (DoS) attacks. In particular, they are quite resistant to any middleware that will lower their ability to tackle these issues. This concern naturally eliminates the use of peer-to-peer mechanisms, which fundamentally change the client-server interaction and security model. Matrix thus uses the same client-server architecture preferred by game developers. This allows developers to reuse existing anti-cheating and anti-DoS mechanisms.

Separation of Concerns : To make developing distributed games easier, Matrix provides a clean "separation of concerns" programming model where Matrix handles the distributed computing aspects of a game such as consistency, scalability, resource provisioning and fault-tolerance, leaving the MMOG developer to focus on the core game logic. The Matrix API is presented in more detail in Section 4.

Support Multiple Gaming Platforms : Game developers frequently develop games for multiple gaming platforms and having to write new Matrix routines for each platform would hinder adoption. As shown in Section 4, Matrix's API does not require any new Matrix-specific routines for a new platform.

Simplicity : Building and debugging a large distributed system is a tricky endeavor as such systems are difficult to debug. As such, Matrix intentionally uses the simplest possible algorithms and APIs. The simple algorithms allow Matrix to be easier to debug and maintain, while the API allows existing games to be quickly and easily modified for use with Matrix.

2.2 Supports Game Requirements

The second key criteria was that Matrix must support the performance requirements of massively multiplayer games. In particular Matrix must provide:

Low Response Latency : Response latency, the time between a game client's action and the observed reaction in the game world, is one of the crucial factors influencing a player's overall gaming experience. Matrix ensures that this latency is as low as possible by not unnecessarily buffering packets and by using an O(1) route lookup mechanism to determine where to send packets (explained further in Section 3.2.4).

Localized Consistency : It is vital that Matrix ensure that the MMOG players are consistent with nearby objects, thus allowing these players to correctly interact with these objects. Since MMOGs are nearly decomposable, it is unnecessary to provide global consistency. Matrix thus provides fast, yet effective, localized consistency mechanisms (explained further in Section 3.1).

Automatically Handle Load Spikes : Load spikes are caused when a large number of players simultaneously decide to visit the same location in an MMOG. It is important that Matrix is automatically able to handle these load spikes without a significant increase in latency. It would also be useful, to conserve resources, if Matrix is able to dynamically change its server usage based on the current game load. We describe how we achieve this in Section 3.2.3.

2.3 Scope of This Work

In this paper, we focus primarily on the Matrix architecture and API along with mechanisms to provide low latency localized consistency. We do not address many other issues that will be needed for a deployed infrastructure such as smart multicast mechanisms to eliminate duplicate packets [14] and service-oriented components, such as user management, authentication and fraud detection. We assume the availability of server resources and mechanisms to find these resources. For the evaluation, we assume that resources can be totally dedicated to a specific MMOG. In practice, server instantiation mechanisms [20] and virtualization techniques [28] would be used to dynamically create and run multiple MMOGs on the same server. We also do not describe any new mechanisms to make Matrix resilient to hardware or software failures. Instead, we plan to reuse existing technology, like failover switches and redundant servers coupled with heartbeat and recovery mechanisms. We also do not present any new anti-cheating or anti-DoS mechanisms in this paper.

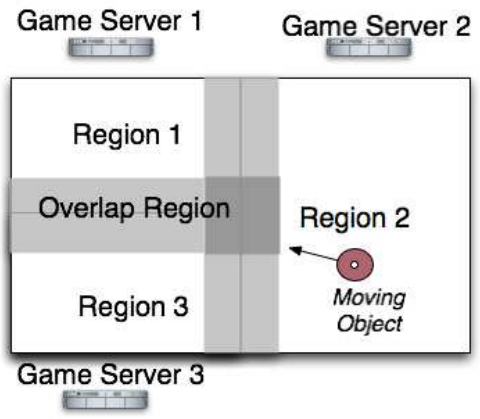


Figure 1: Overlap Region between 3 Matrix Servers

3 Matrix Design and Implementation

In this section, we describe Matrix's design and implementation, focusing primarily on the overall architecture and major technology components. In Section 3.6, we present an example demonstrating how Matrix uses its various pieces to support a game.

3.1 Providing Localized Consistency

To build an easy to use localized consistency mechanism, we observed that all games have some notion of geometric space that allows distances between game objects to be computed using a game-specific distance metric. If Matrix was aware of an individual game's *spatial coordinates* and its *radius of visibility* (the range over which local consistency is typically required), it could confine the propagation of any game state update to an easily computable region, without having to maintain game-specific relationship trees or other data structures. Matrix uses this insight to require game developers to merely forward all game packets, appropriately tagged with the spatial coordinates (in the game world) of the packet's origin and destination, to the local Matrix server. Matrix uses these spatial tags, together with the game's radius of visibility, to route these packets to the other game servers that manage objects within this radius of visibility (and thus need to maintain consistency with the original game server).

Matrix assigns unique portions of the MMOG's spatial map to different servers. Each server is only responsible for clients located within its assigned partition. Formally, Matrix partitions the overall space Z of an MMOG into N non-overlapping partitions, $\{P_1, P_2, \ldots, P_N\}$, and assigns each partition P_i to a distinct server S_i. To handle load spikes, the number of servers N, and the specific partition managed by any server S_i can change dynamically.

Because games have a non-zero radius of visibility, changes in the MMOG state at any point, σ_i, handled by server S_i, that is within the radius of visibility of a client located on server S_j, must be consistently applied at both servers S_i and S_j. In general, given a spatial partition and a radius of visibility R, every point σ in Z has a set of servers associated with it, called the *consistency set* of σ or $C(\sigma)$. This set contains all the servers whose partitions overlap the circle (or sphere) of radius R centered at σ and therefore need to be aware of any update or activity at σ. If $d(x,y)$ represents the distance-metric between points x and y,

$$C(\sigma \in P_i) = \{S_j | j \neq i \ \wedge \ \exists \sigma' \in P_j \ s.t. \ d(\sigma, \sigma') \leq R\} \tag{1}$$

From Equation 1, we observe that if R is infinite, *all* updates must be globally propagated, making localized consistency impossible. However, if R is small compared to the size of partition P_i, most of the interior points of P_i will have empty consistency sets. Only the relatively small number of periphery points, whose $C(\sigma) \neq \emptyset$ (i.e, whose radius of visibility extends into adjoining partitions) will require consistency to be maintained between servers. Games usually have limited player visibility radii and Matrix efficiently utilizes this sparseness by forming groups, called "overlap regions", of all points that have identical non-empty consistency sets (shown in Figure 1).

Intuitively, an overlap region denotes a portion of the map, such that an update at any point in that overlap region requires all the servers in that overlap region to be informed of the update. Overlap regions allow Matrix servers to quickly determine the consistency set for any game packet they receive by merely doing a table lookup (of the set of overlap regions). When the overlap region is as large as the entire map, Matrix effectively becomes a distributed shared memory (DSM) system [2]. Unlike DSMs, we can constrain the overlap regions for MMOGs (because they are nearly decomposable) using geometric distances thereby satisfying scalability and latency constraints that proved to be elusive for DSMs.

Matrix assumes that most players in a game have the same radius of visibility. The Matrix API does allow game servers to specify different visibility radii for exceptions, and internally creates distinct sets of overlap regions, each for a different R. We decided to use overlap regions instead of other geometric data structures, like spanners [4], to determine the consistency set of any object as overlap regions do not require costly (in terms of latency) hop-by-hop lookups and they work well even when the map space changes dynamically (which happens during splits and reclamations).

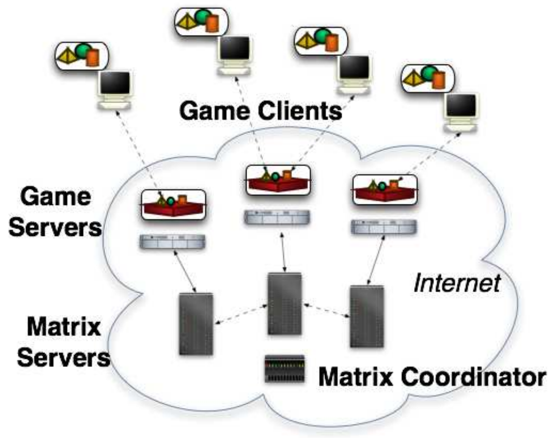


Figure 2: Matrix Architecture

3.2 Matrix Architecture

Figure 2 shows the Matrix architecture, that satisfies the design criteria in Section 2. A MMOG is deployed using Matrix as follows: MMOG developers provide the game clients and game servers, while the Matrix infrastructure provides the Matrix servers and a Matrix coordinator (MC). The architectural components interact as follows:

3.2.1 Game Clients

The clients are used by game players to play the MMOG. Each client interacts with a game server and provides it with updates on the player's activity and receives updates on nearby activity. Game clients must be able to switch servers *dynamically* as the MMOG may be on multiple servers, each handling a unique portion of the MMOG world. The client is informed of these switches by its current game server and is unaware of the presence of an intermediate Matrix server.

3.2.2 Game Servers

The game server is the software that stores the state of the game world and coordinates the activity of the players in the game. In most commercial games, they are also the only point of contact between game clients and the game world to protect against cheating and unauthorized collusion; problems that are particularly acute in multiplayer games. The game server must be designed for use in a multiserver environment. In particular, it must identify players using globally unique IDs (such as callsigns) instead of locally generated IDs. Game servers are usually located on the same physical machine as a Matrix server (to minimize the network latency). In our current implementation, the Matrix server is a separate process from the game server. In the future, we may compile the Matrix server into the game server (as a separate library) to improve performance.

When a game server starts, it sends Matrix the visibility radius of clients in the game (to allow overlap regions to be correctly computed). The game server then forwards all client packets (after spatially tagging them) to its Matrix server for further processing. The game server also periodically reports its current load to Matrix. If the server is overloaded, Matrix splits the game world between the overloaded server and a newly created game server and informs both the new and overloaded game servers of their new map ranges. The overloaded game server then forwards all game specific state (e.g., map objects such as trees, buildings etc.) to the new game server via Matrix. Finally, the overloaded game server redirects any clients (and their corresponding state) that are not in its new map range to the appropriate game server (Matrix provides the identity of the appropriate game server). Moving these clients to other game servers decreases the load on the overloaded game server. However, if it is still overloaded, Matrix splits the still overloaded game server again until it has shed enough load.

3.2.3 Matrix Servers

Matrix servers, the heart of our distributed middleware, provide the necessary consistency, reliability and latency semantics for MMOGs. Each Matrix server is aware of the map range currently managed by the game server connected to it. On receiving spatially tagged game packets from its game server, the Matrix server checks its overlap tables, provided by the MC, to see if any peer Matrix servers are within that packet's consistency set. If so, the packet is forwarded to these peer servers which then forward the packet, after verifying the packet's range, to their own game servers for processing. Because Matrix handles packet routing, individual game servers do not need to know about other game servers serving the MMOG.

Matrix splits map partitions using purely local decisions to improve scalability and minimize latency. On detecting that its game server is overloaded (through explicit load messages from the game server or via system performance measurements), a Matrix server first checks, using some non-Matrix external entity, for an available Matrix server. If a server is available, it splits its current map, keeping control of a sub-portion of the map, while transferring responsibility for the remaining portion to a new Matrix server. Currently, Matrix uses a simple "split-to-left" splitting technique where each map is split into two equal pieces with the left piece handed off to the new server. This simple algorithm still provides good performance as shown in Section 5.

The new Matrix server then creates a new game server and orchestrates the transfer of the global state needed to play the game, from the original overloaded game server to this newly-created game server. The overloaded game server then switches game clients to this new server to ease its load. The amount of state associated with switching game clients is usually minimal (shown in Figure 6 for the games used to test Matrix) and Section 3.4 details the mechanisms used to transfer client state. Newly started game servers also need to obtain the static state of the game, like the map textures, that can be hundreds of megabytes in size. However, because this state is static, it can be pre-cached on all new servers, requiring only pointers to the cached state to be sent.

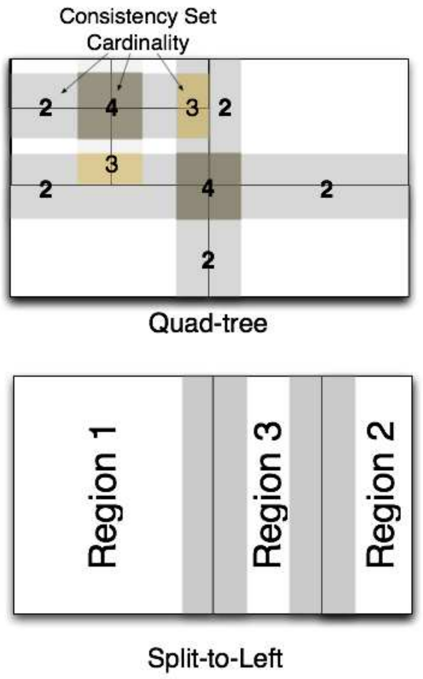
The Matrix server that performed the split becomes the parent of the newly created Matrix server. Whenever a Matrix server detects that its game server is underutilized (again, through explicit load notifications or via system performance measurements), it first checks if it has any children. If it does and if their load levels are low enough, the parent Matrix server reclaims the partition and game state held by the child. All the game clients on the child's game server are transfered to the parent's game server, after which the child Matrix server and game server are removed from the game and returned to the resource pool. Section 3.5 describes how we prevent oscillations and ensure stability in the splitting / reclamation process.

3.2.4 Matrix Coordinator (MC)

The MC is used to create the overlap tables used by Matrix servers to route spatially tagged packets. Whenever a new Matrix server is used for the game, it informs the MC of the current map range and radius of visibility of its game server. The MC then computes the overlap regions for all the Matrix servers in the game using geometric algorithms to calculate bounding boxes between spatial regions – a particularly easy computation, using well known axis-aligned bounding box computation algorithms, if the map partitions are rectangular in shape. The MC then informs each Matrix server of their overlap regions along with the set, $C(\sigma)$, of Matrix servers that should be informed about an event in that region. The MC recomputes and redistributes overlap regions every time a new Matrix server is used or whenever an existing Matrix server is reclaimed (the MC is informed of the new map ranges whenever reclamations occur).

We used a central MC to minimize the latency of the packet forwarding process. In the common case where players are only interacting with nearby objects, each Matrix server can do an instant O(1) lookup to determine the consistency set for any game packet using the overlap regions provided by the MC. Even in uncommon cases involving non-proximal interactions, the Matrix server can consult the MC to determine the consistency set for that particular interaction. Matrix could use alternate lookup methods (such as DHTs [25]), but that would result in increased latency (e.g., DHT schemes usually need $O(log(N))$ lookups for N Matrix servers). Although a centralized approach can lead to performance bottlenecks, the MC is only used when the MMOG world partitioning changes due to splits or reclamations (which should occur infrequently for a stable game). This centralized approach can scale to large server populations as the MC is not in the latency-critical packet forwarding path (except for the rare non-proximal interactions). Experimental evidence, presented in Section 5.3.3, shows that the MC does not become a bottleneck even when Matrix scales. The MC can also be made reliable using well understood replication techniques.

Another possible solution would be to eliminate the MC and have every Matrix server run a routing protocol, similar to RIP [16] or OSPF [17], to maintain a completely up-to-date routing table containing entries for every other Matrix server. However, this adds substantial programming and debugging overhead to the Matrix infrastructure as these distributed protocols are difficult to develop and debug. In addition, the use of an MC, which has knowledge of the entire Matrix infrastructure and overlap regions, allows Matrix to perform better repairs in cases where a Matrix server dies. In summary, the use of an MC provides a simple and easily verified solution that allows Matrix to perform O(1) routing and perform efficient global repairs without incurring any significant performance overheads.



This figure shows 2 different ways to split a Matrix server. The quad tree split (on top), where partitions are split alternately along the x and y axis, results in areas of the map having a large consistency set. The bottom figure show the split-to-left scheme, where partitions are only split along the x-axis. The overlap regions are shown shaded.

Figure 3: Different Splitting Techniques

3.3 Map Partitioning (Splitting) Strategies

A key challenge was the strategy used for splitting the map partition managed by an overloaded server. Ideally, we would like to form balanced sub-partitions, such that the parent's workload is equally divided between itself and its child. A simple "divide the map by half" splitting technique (as used in Section 3.6) may not satisfy this requirement as the load may still be concentrated on one of the servers.

In general, balanced partitioning can only be performed by the game server, because it alone is aware of the precise spatial skews in the workload. However, this could be a difficult task for game developers to undertake. Matrix therefore provides a simple, default, splitting algorithm while allowing game developers to specify more complex and game-specific splitting mechanisms via the Matrix API if they so choose. Our simple algorithm is to split the current map into half along the y-axis and is adequate for situations where clients are evenly distributed among the two halves. Clearly, it won't be as efficient (needing multiple sub-splits) for very skewed workloads such as a hotspot caused by a large number of clients in a small region of the map. However, even in these cases, Matrix's performance (shown in Section 5) is still acceptable.

To preserve Matrix's ability to scale while keeping latencies low, the splitting algorithm must ensure that the consistency set associated with any particular point doesn't become very large. Some seemingly simple splitting strategies do not, however, satisfy this "bounded consistency set" (or *BCS*) property. For example, Figure 3 shows a quad tree based splitting strategy, where partitions are alternately split along the horizontal and vertical axes. As the diagram illustrates, after multiple levels of splitting, the consistency set, $C(\sigma)$, for a point close to the center of the world map can easily be as large as $\{S_1, S_2, S_3, S_4\}$, even though the radius of visibility R is smaller than the size of the smallest partition.

In our current Matrix implementation, we employ a "split-to-left" strategy, where each parent server splits its partition (in 2D or 3D) along the x-axis (chosen arbitrarily), and then gives the left half of the split to the child server, while retaining the right half of the split. Figure 3 demonstrates this splitting mechanism, where server S_1 first splits and creates server S_2. Subsequently, S_1 splits again to create a new server S_3. This strategy creates rectangular overlap regions, of breadth $2 * R$, that are easily describable (bottom left and top right coordinates are sufficient) and satisfies the following property.

Property 1 *If the width Δ of any partition exceeds twice the radius of visibility ($2 * R$), the 'split-to-left' strategy satisfies the BCS property, and the cardinality of the consistency set for any point σ in the map is at most 1. If $R < \Delta < 2 * R$, the BCS property is satisfied and the cardinality of the consistency set for any point σ is at most* 2. *Otherwise the cardinality is* $2 * \lceil \frac{R}{\Delta} \rceil$

The above principle shows the central role that R plays in the scalability of Matrix. When the size of a new split is smaller than R, the consistency set for any point σ on the new server will extend beyond its immediate neighbors. Hence, R determines when the consistency set will contain inefficient non-neighboring partitions. It also determines how many clients will be in the overlap regions as shown in Section 5.4. The width of a partition at depth D is given by $\frac{L}{2^D}$ (because each split divides the current map into half), where L denotes the width of the entire MMOG map. Accordingly, the maximum depth D_{max} to which Matrix can split before the partition width becomes smaller than R is given by:

$$D_{max} = \lfloor \log_2 (\frac{L}{R}) \rfloor. \qquad (2)$$

As long as Equation 2 is satisfied, the total number of overlap regions will not exceed N $(= 2^D)$, the number of deployed servers. Experimental evidence with various MMOGs, shown in Section 5, suggests that, while the split-to-left mechanisms is somewhat inefficient in terms of the number of servers used, it is adequate in responding to load changes. The development of more optimal spatial partitioning algorithms remains an open research problem for future versions of Matrix.

3.4 Consistent Low Latency Handoffs

When a client switches from server GS_i to GS_j, its state must also be transfered. Without this state transfer, GS_j might reject packets from redirected clients thinking that they are invalid or malicious. Matrix uses a combination of client-driven and proactive state transfer mechanisms to transfer this state as quickly as possible.

In the client-driven approach, if a game server GS_j receives game packets from a client that it is not currently handling and doesn't have sufficiently fresh state (for a game-specific parameter), it buffers the packets and requests the client's state from Matrix. The local Matrix server will route this query to nearby Matrix servers that it shares an overlap region with, which will in turn pass this request up to their game servers. This localized query approach is effective, because clients usually only enter a partition from neighboring partitions. When the dynamic state of the client is located on a neighboring game server, it is relayed back to GS_j by Matrix. On certain rare occasions (e.g., when a client teleports from one part of the map to another), this localized query may be ineffective, and the local Matrix server has to broadcast a state-retrieval query, via the MC, to all servers.

This approach provides eventual consistency at the cost of large per-client buffers and significant latency (often $O(secs)$), as the client is effectively frozen until its state is retrieved. To reduce this latency, Matrix also uses pro-active (Matrix-driven) state transfers. Whenever a Matrix server discovers that a client needs to be switched (game servers contact Matrix to discover where to switch clients), it proactively requests the client's state from its game server and forwards it to the target game server. This approach eliminates most of the state retrieval latency even for teleportation-like events.

3.5 Preventing Oscillations and Ensuring Stability

As stated earlier, splits and reclamations are triggered by server load-based thresholds. To prevent system oscillations, the high (splitting) and low (reclamation) thresholds of the load metric must satisfy the following property:

$$\forall\{x_1,x_2\}, \quad ((L(x_1) < T_l) \quad \bigwedge \quad (L(x_2) < T_l)) \implies (L(x_1+x_2) < T_h) \tag{3}$$

where L is the load function, x_1 and x_2 are measured raw load values and T_l and T_h are the low and high thresholds respectively.

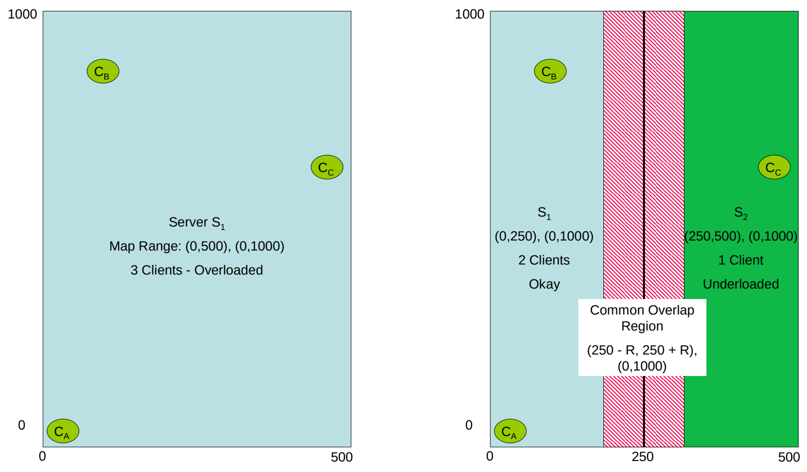
Equation 3 ensures that reclaimed nodes, without sudden load changes, will not immediately split after being reclaimed. At runtime, we also use an additional hysteresis function to prevent oscillations caused by loads fluctuating close to the thresholds. To ensure that the split / reclaim process is stable, only parents are allowed to reclaim children and no server can be both splitting and reclaiming (or in the process of being reclaimed) at the same time.

3.6 Matrix in Operation

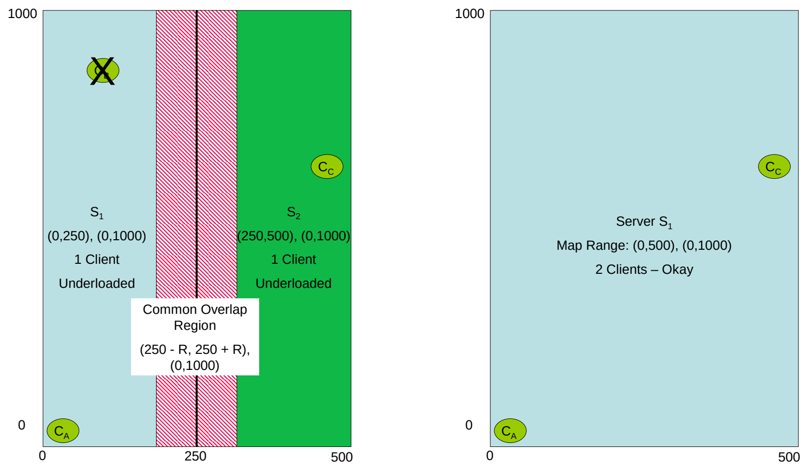
We illustrate how the four Matrix components interact through a simple example using a 2-D game with a rectangular map of dimensions 500X1000 units. Assume that the radius of visibility for all clients is $R = 100$ units and that a game server becomes overloaded when it has 3 or more clients and underloaded when it has 1 or less clients. All partitions in this example will be rectangular in shape and represented by their bottom-left and top-right coordinates. Thus, $\{(0,0),(500,1000)\}$ represents the entire game space. A Matrix server (S_1) starts the initial game server (GS_1) and is informed by GS_1 that it is responsible for the entire ($\{(0,0),(500,1000)\}$) space with a visibility radius of R. S_1 forwards this information to the MC which does nothing as there are no overlap regions at this point in time. Clients c_A and c_B then connect to server GS_1 and start playing the game. GS_1 forwards every game packet, tagged with their spatial coordinates, to S_1, which simply discards them as each packet's consistency set is null.

A new client, c_C, now joins the game. Let the instantaneous locations of the three clients be $(20,20)$, $(150,900)$ and $(450,700)$ respectively (Figure 4a). At this point, server GS_1 becomes overloaded (because it is currently serving 3 clients) and informs S_1 of that fact. S_1 starts the splitting process and informs the new child Matrix server (S_2), found via an external discovery mechanism, that it is going to be used for the game. S_2 then starts a new game server GS_2 while S_1 decides how to split the game map between itself and S_2. In this case, the map is split in half, leaving S_1 to manage the partition $\{(0,0),(250,1000)\}$, while assigning S_2 the partition $\{(250,0),(500,1000)\}$. S_1 informs GS_1 of its reduced spatial responsibility (S_2 also informs GS_2 of its spatial responsibility), and requests from GS_1 the game-specific objects and state for the portion $\{(250,0),(500,1000)\}$ that GS_1 was previously managing. This state (which remains opaque to Matrix) is forwarded to S_2, which relays it to GS_2, thus initializing GS_2 with the state needed to manage its region of the world. The MC is informed of the new map ranges for both S_1 and S_2 and calculates overlap regions for them. In this case, S_1's overlap region is $\{(250-R,0),(250+R,1000)\}$ with a consistency set of $\{S_2\}$. S_2 has the same overlap region with a consistency set of $\{S_1\}$ (Figure 4b). Anytime a client enters this region of space, the client's actions is forwarded to the other Matrix server.

GS_1 checks if any of its current clients need to be switched to GS_2 and discovers that c_C needs to be switched. GS_1 sends a request to S_1 stating that client c_C at position $(450,700)$ needs to be switched. S_1 consults its internal routing tables, discovers that S_2 is responsible for c_C's map range, and tells GS_1 to switch



a) Server S_1 is overloaded b) After Server S_1 Splits to Server S_2



c) Client C_b leaves d) After Server S_1 Reclaims Server S_2

Figure 4: Matrix in Operation Example

c_C to GS_2. GS_1 then sends the state associated with c_C to GS_2 (via S_1) and tells c_C to connect to GS_2. c_C connects to GS_2 and continues playing the game with minimal interruptions (as GS_2 has all the state associated with c_C).

GS_2 now has only 1 client and informs S_2 that it is underloaded. However, because S_2 has no children, no further action is taken. After some time, c_B leaves the game causing GS_1 to also become underloaded (Figure 4c). S_1, on receiving this information, queries its child (S_2) for its current load. S_2 queries GS_2 for its current load and replies to S_1 that it is currently underloaded. S_1 proceeds to reclaim S_2 (because the loads on S_1 and S_2 satisfy the requirements explained in Section 3.5) by reclaiming the map range $\{(250,0),(500,1000)\}$ from S_2 and informing GS_1 that its new map range is $\{(0,0),(500,1000)\}$. S_1 then sends GS_1 all the game state associated with the reclaimed partition (received from GS_2 via S_2). GS_2 switches its client, c_C, to GS_1 using the process mentioned above. The MC is informed of the new map range of S_1 and that S_2 has been reclaimed. Finally, S_2 kills GS_2 and puts itself back into the available resources pool (Figure 4d).

4 Matrix API: Specification and Use

In this section, we describe the API that makes it easy for game developers to use Matrix along with our implementation experience in modifying games to use Matrix.

4.1 Game Client API

Game clients have no direct knowledge of Matrix and only need to be able to switch game servers dy-

```
int matrix_connect          (char * servername, int port);
int handle_matrix_commands  (int fd);
int send_map_range          (int fd, map_t range);
int receive_map_range       (int fd, map_t &range);
int send_visible_radius     (int fd, radius_t radius);
int receive_overlap         (int fd, overlap_t &overlap);
int send_load               (int sockfd, int load);
int client_switch_query     (int fd, client_t split);
int client_switch_reply     (int fd, client_t &split);
int send_state              (int fd, static_t static);
int receive_state           (int fd, static_t &static);
// radius is used for packets with non standard  visibility radii
int send_spatial_packet     (int fd, pos_t src, dest, char *msg, int msg_len,
                             radius_t radius);
int specify_map_split_point (int fd, split_t split);
```

Figure 5: API used between Matrix and game servers

namically while playing the game (either directly or via a proxy). Game clients designed for multiserver games should already be able to do this.

4.2 Game Server API & Modifications

Game servers need to implement the functionality, shown in Figure 5, to support the Matrix API. In particular, the game server must connect to Matrix on startup (a Matrix networking library is provided). It must also identify the coordinate system used by the game and report it, along with the size of the game world and the common visibility radius (determined by the game developer), to the Matrix server when requested. Finally, it must send anything that affects the game world to Matrix tagged with the coordinates of the point in the game world that is affected (the Matrix networking library provides routines to do this). To send non-proximal packets, the game server must set the `dest` argument of `send_spatial_packet` accordingly. Global broadcast packets can also be sent by setting the optional `overlap` argument in `send_spatial_packet` to a larger overlap radius just for that packet.

To report the current game load, developers can either write a game-specific load monitor(with appropriate thresholds that satisfy Equation 3) or let Matrix calculate the load from system parameters such as memory usage or network throughput. When using system parameters, developers can either specify the thresholds (wrong thresholds can cause bad game performance) or use Matrix-supplied values.

Whenever an object moves in the game world, the game server must check if the object is still within its map range. If it isn't, the game server must tell the object to switch to the correct game server (Matrix is consulted to determine the correct server) or switch the object itself (if it's a non-player object such as a monster). Matrix will handle the state transfer necessary for moving the object as explained in Section 3.4. The hardest game server modifications are, the routines to exchange state with Matrix as this usually requires packing various internal data structures into a packet stream to send to Matrix. These routines also usually need to be highly optimized to achieve good performance. Game servers must also be able to receive this packet stream from Matrix and recreate the state after verifying that it is fresh (by comparing timestamps). Before accepting clients, game servers must check if they are new players (possibly by asking a 3rd party registration service or by accepting a cookie from the client). If the client isn't new, the game server must ensure that it has sufficiently fresh state for that client. Otherwise, it must request the state from Matrix and prevent the client from entering the game until the state has arrived. The time period after which state is considered stale is game specific and set by the game developer.

4.3 Implementation Experience

Our experience indicated that game servers can be modified to work with Matrix in a very short amount of time. For this paper, three real games (described further in Section 5) were modified to use Matrix. It took us between 8 to 16 hours per game to integrate the game with Matrix. However, this excluded the time needed to understand these games (to determine how state was kept in the server, what the spatial coordinates of the game were, etc.). It also excluded the time needed to make these games multiserver compatible. These games used non-unique local ids (like array indices) to identify individual players as they were designed for single server use. Hence, even though Matrix integration was easy, the games were unable to handle packets from other servers due to id collisions. It required about 1 week to convert each game to use globally unique player ids, like callsigns, before they worked properly with Matrix. This is not a Matrix limitation: any multi-server game would require globally unique ids.

The 8 to 16 hour modification times are thus indicative of how long game developers, who understand their multiserver-friendly games well, need to implement the Matrix API calls (shown in Figure 5) and support the requirements outlined in Section 4.2. We feel confident that these times are low enough that Matrix integration is not a bottleneck for game developers.

5 Performance Evaluation

In this section, we evaluate the performance of Matrix. Our evaluation goals were to show that even with its simplicity, Matrix is able to both satisfy real game players and achieve good system-level performance. We answer the first question through a small user study of Matrix with a real game (shown in Section 5.2). To answer the second question, we performed system level measurements (shown in in Section 5.3).

5.1 Test Games

Unfortunately, we were unable to obtain the source code for any real MMOG as game developers were unwilling to release their proprietary game sources. Thus, we resorted to using three open source games that are representative of some commonly encountered MMOGs:

- *Bzflag* [19] (version 1.72g2) is a popular open-source tank game that allows up to 200 players in individual tanks to fight against one another (individually or as teams) on a single server using a variety of weapons. Bzflag, initially released in 1993, is still being actively played and developed.
- *Quake2 by Id Software* [11] (version 3.20 released under the GPL license in 2002) was released in 1998 and is one of the best selling first-person shooting games ever with over 1.2 million copies sold worldwide. It features a heavily armed soldier fighting against hordes of alien enemies. Its multiplayer component allows up to 256 players to compete against each other, on a single server, in "deathmatch" style games where each player fights against every other player.
- *Daimonin* [27] (version 0.95b2) is one of the few open source role-playing games. It is still in early development, but already allows multiple players to enter a world map (on a single server) and go on quests, find treasure, defeat monsters and interact with other players. In Daimonin, like other role-playing games, the goal is to gradually increase the ability of one's character through various adventures and interactions with other game players.

In general, first person shooting games have stricter latency requirements and send more update packets, while role-playing games store a lot more player state (as players can engage in many more activities). This is shown in Figure 6 which lists the actual runtime parameters that were measured for each of the three games.

We modified each game to work with Matrix. We also developed robot clients for each game that were able to move freely in the game world and could be switched to different servers by Matrix. The distributed implementation of Bzflag is fully functional and human players of Bzflag were used for the user study shown in Section 5.2. Even though Quake2 and Daimonin were modified to use a globally unique namespace, we have not extensively tested their game clients with human players.

5.2 User Study Evaluation

In this section, we present the results of a small user study, involving real game players, that was designed to answer the following questions:

1. How do players perceive the movement of their in-game avatars under different loads in a multi-server Matrix-enabled game?

	Bzflag	Quake2	Daimonin
Map Range	800x800	Highly Variable[a]	24x24 tiles[b]
Visibility Radius	200	400	24
Max Players[c]	200	256	1024
Spatial Packet Size (Bytes)	98	60	18 - 40
Packet Send Rate (packets/s)	0 - 60	0 - 200	0 - 3
Client Velocity (map coords/s)[d]	0 - 40	0 - 40	0 - 5
Dynamic State Size (KBytes)[e]	394	182	2198

[a]depends totally on the map being played
[b]The world is built using multiple 24x24 map "tiles"
[c]hard player limits before the game was modified to use Matrix
[d]measured maximum and minimum client movement rates
[e]amount of player state that must be sent when switching servers

Figure 6: Measured Runtime Values of the Test Games.

2. How do players perceive the movements of other players' avatars under different loads in a multi-server Matrix-enabled game?

3. Is the Matrix splitting / reclamation mechanism noticeable to game players?

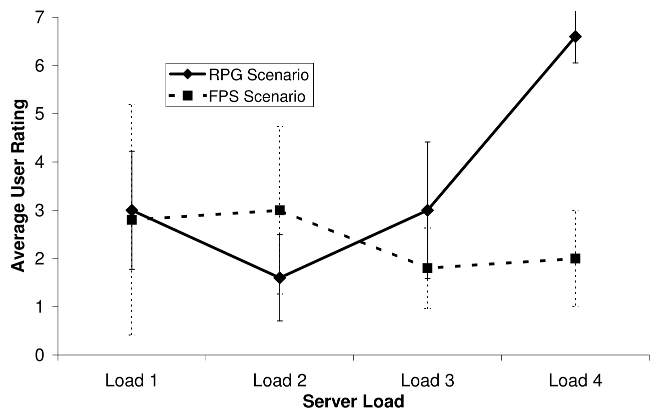
For this experiment, we used a version of Bzflag that allowed players to move freely in the game world and fire at other tanks in the game. We disabled the ability to actually kill tanks (and to be killed) to prevent players' tanks from dying during the experiment. We used two different scenarios for this experiment. The first scenario used 4 different loads ranging from 40 to 160 robot tanks, that had update rates of 100ms each (each client updates the server every 100ms and receives updates from server as necessary), in increments of 40 to simulate a first-person shooting (FPS) game. The second scenario used 4 different loads ranging from 100 robot tanks to 400 robot tanks, that had update rates of 1s each, in increments of 100 to simulate a role-playing game (RPG).

For each scenario, the highest load level was chosen so as to subject the system to extreme conditions. We started the test by showing the players the performance of a lightly-loaded game running on Matrix as well as a heavily-loaded game running on Matrix. In both cases, Matrix was using only a single server so the game experienced no overhead. We then varied the configurations and asked the players to rate the performance of a game at each configuration. The ratings were from 1 (lightly loaded / no effect on game performance) to 7 (heavily loaded / game is horrible to play) as normalized by our initial demonstration. Five people participated in this study.

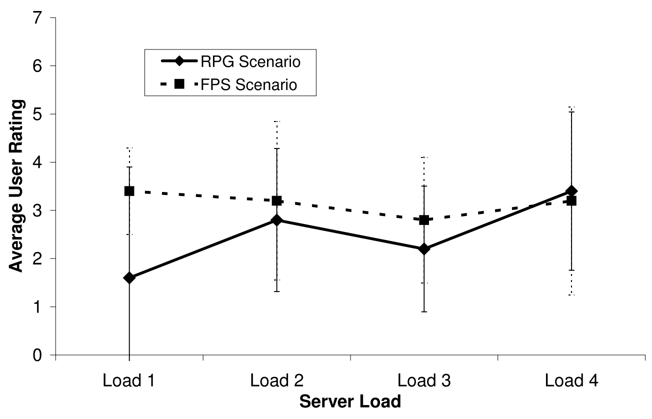
To answer the first question, we statically split the game between two Matrix servers and asked the players to travel around the perimeter of the game world under various load conditions. This movement pattern guaranteed that participants crossed the server boundary two times, requiring Matrix to switch the player two times between servers. After traversing the entire perimeter of the game world, we asked participants to rate the performance of the game.

To answer the second question, we created two special tanks in the game. One tank was positioned at a strategic point in the game where it could observe objects moving across the server boundary. All the players were asked to look at the display for this tank (to ensure that the scenario for this test was common across all players). The other tank was controlled by an experimenter and it proceeded to cross the Matrix server boundary at least three times (switching servers each time) at a point visible to the other tank. After observing these server crossings, the players had to rate the performance of the game; in particular whether the movement of the experimenter's tank was smooth. Figure 7 shows the average user perception for the first and second questions for the various scenarios and load levels.

To answer the third question, players were asked to monitor the movements of the experimenter's tank



(a) Personal Game Experience



(b) Observed Switching Time

The two Figures show the average user rating by test game players when using Matrix for two different scenarios. The loads (load 1, load 2, load 3 and load 4) for the RPG Scenario were 100, 200, 300 and 400 tanks respectively with 1s update rates and 40, 80, 120 and 160 tanks for the FPS Scenario with 100ms update rates. The average user rating was a linear score that ranged from 1 (lightly loaded / no effect on game performance) to 7 (heavily loaded / game is horrible to play).

Figure 7: Results of User Study

(by looking at the experimenter's display) while robot tanks were repeatedly inserted and removed from the game. For this experiment, the game was initially run on just one server with a high load threshold of 50 tanks and a low threshold of 10 tanks. After some time, 120 FPS robot tanks (100ms update rate) were introduced into the game all at once. This caused Matrix to split the game across two servers and switch tanks accordingly. Some time later, the 120 robot tanks were removed all at once causing Matrix to reclaim the entire game onto a single server. This process was repeated multiple times. During all this, the experimenter's tank was traversing around the game world and shifted servers several times as a result of splits / reclamations. After allowing the players to observe five server splits and reclamations, they were asked to rate the performance of the game. We then repeated this experiment with 300 RPG tanks (1s update rate). The average user rating was 3 (std. = 1.73) for the FPS scenario and 3.4 (std. = 2.07) for the RPG scenario.

Overall, the user study suggest that players found the latency, including the client switching latency, of the Matrix-enabled game to be acceptable except in the extreme cases where the servers were overloaded. The latency when Matrix split and reclaimed servers in response to load was also acceptable. An interesting side effect of this study was that we could observe how users perceive the same events very differently. Our average scores and standard deviations were raised greatly because of only one user who had a very different perception from the other four players – even for experiments where the players observed exactly the same game behavior (they were all looking at the same screen). For example, if we discount this user's scores, the average user rating for the third question becomes 2.25 and 2.5 for the RPG and FPS scenarios respectively. More importantly, the standard deviations dropped to 0.5 and 0.58 (from 1.73 and 2.07) respectively. This effect was seen for the results in Figure 7 as well. However, even with this user's results taken into account, Matrix still performed well.

5.3 System Level Measurements

In this section, we validate the system-level performance of Matrix. We analyze the ability of Matrix to adapt to hotspots in Section 5.3.2 and measure its overhead in Section 5.3.3.

5.3.1 Experimental Platform

For this evaluation, we implemented all the core components of Matrix, except for the reliability component. The current version of Matrix, written in C, exports the API shown in Figure 5, uses a MC to obtain routing and overlap information and performs the localized consistency, state transfer and on-demand protocols explained in Section 3. The experiments were performed on HP Omnibook 6000 notebooks with 256 MB of memory, a 20 GB hard disk and a 1 GHz Mobile Pentium 3 processor running Redhat 7.2 on a 2.4.18 linux kernel.

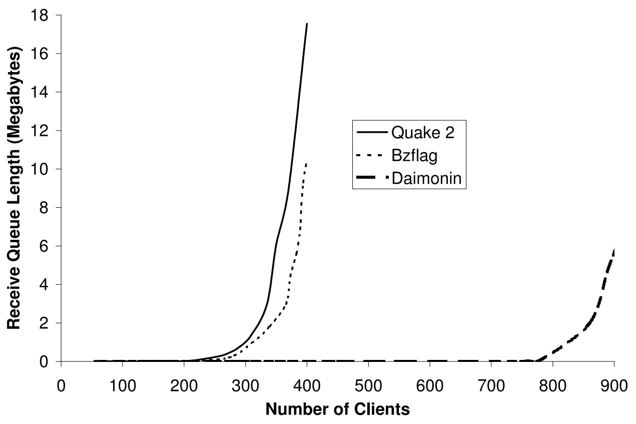


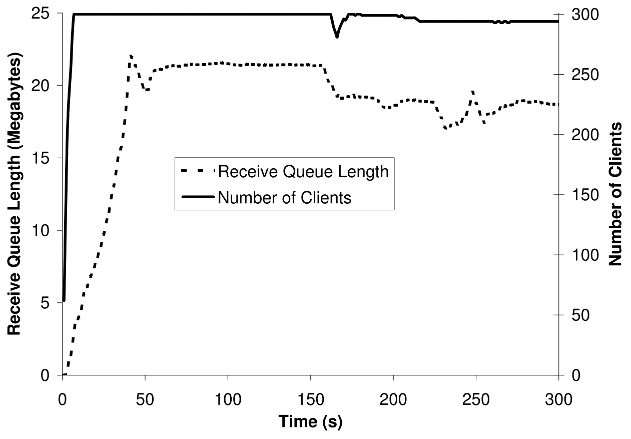
Figure 8: Break Point of a Single Server

For all experiments, we used the number of players in the game as the load metric. We conducted load tests, for all 3 games, on a single server. From the results (shown in Figure 8), we set the high (splitting) load threshold (Section 3.5) for Bzflag and Quake2 to 300 clients each and Daimonin to 800 clients. We set the low (reclamation) thresholds at 200 clients for Bzflag and Quake2 and at 600 clients for Daimonin.

5.3.2 Adapting to Hotspots

In this section, we compare the performance of Matrix against a static partitioning scheme (because these schemes are used by commercial MMOGs) in cases where the load changes dynamically (such as when hotspots occur). We changed the load by using different client population distributions (because the load metric was the number of players in the game). We performed the measurements using a small number of servers. We used four notebooks that had no resource limits imposed on them as Matrix servers (they also ran the game servers), one notebook as a dedicated MC, and three notebooks to generate clients. For each experiment, we sampled the server queue length and number of clients on each server at 0.1 second intervals. We did not measure client metrics such as client-perceived latency as they are influenced by many factors (such as swapping on the client machine) beyond Matrix's control. Note that the user study (Section 5.2) attempted to quantify client-perceived latency. For this section, we measured the server queue lengths as they can be used to meaningfully compare different partitioning schemes because larger queue lengths will increase the client latencies.

In the first experiment, for Bzflag and Quake2, we created a hotspot, at a particular point on the map,

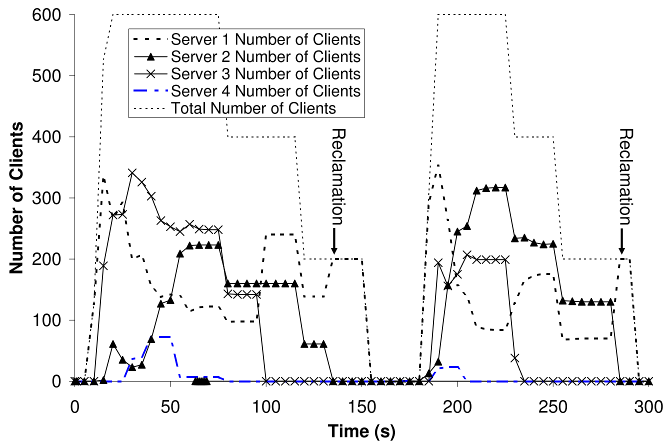


This shows the effect of 300 clients moving to one server simultaneously. The number of clients is read off the right Y-axis

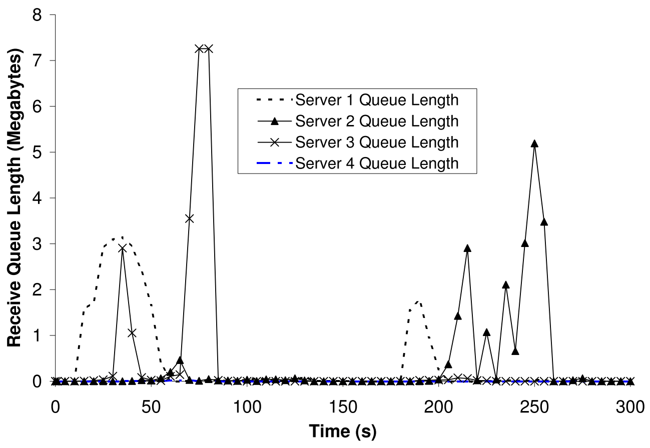
Figure 9: 300 Client Hotspot (Static Partitioning)

caused by 300 clients. For this experiment, we statically partitioned the game between two servers, with each serving half of the game world. Figure 9 shows the number of clients and receive queue length for the server that had the hotspot. From the figure, we see that the server had to handle all of the clients (the number of clients line remains at or very near 300) and its receive queue length increased as a result. The server was able to handle the load (its receive queue length did not increase uncontrollably), but only barely. We were unable to test the static partitioning with larger hotspots as that ended up killing the server with the hotspot. Note that if the clients were randomly distributed, the static partitioning would have done well because it could have shared the load with the other server. However, any a-priori static partitioning will always remain susceptible to unpredictable hotspots that cannot be shared between the static partitions.

Matrix is, however, able to handle even dense hotspots. Figure 10 shows an experiment in which a hotspot of 600 clients, far higher than the static partitioning could handle, was introduced at around the 10 second mark for about 75 seconds, after which the entire hotspot gradually disappeared (indicated by 200 clients disappearing at fixed intervals). The hotspot was reintroduced at a different position in the world at 170 seconds, for about 50 seconds, and then gradually removed. Matrix relieved the initial spike in the receive queue caused by 600 clients joining (shown at time=10 in Figure 10) by spawning server 2 (at time=10) and giving it half the map. However, this did not ease the load as the hotspot was on the map



(a) Number of Clients



(b) Server Queue Length

This Figure shows Matrix responding to hotspots caused by 600 clients. The left graph shows how the total number of clients were shared among the various servers. Note that a server is overloaded when it has 300+ clients. The right graph shows the receive queue length of the various servers. Matrix used up to four server to handle the load caused by the hotspots. However, Matrix reclaimed those extra servers as shown by the reclamation points on the left graph when the load eased. The second reclamation took longer as the child server took longer to become underloaded ($<$ 150 clients).

Figure 10: Hotspot caused by 600 clients

portion retained by server 1. Hence, server 1 spawned another server, server 3, (at time=10) and split its current map with it (servers 1 and 3 have 1/4 of the map each with server 2 having the rest). Server 3's map range contained the hotspot and a large number of clients were switched to it easing server 1's load. However, server 3 now experienced a load spike (at time=60). This process continues recursively until the load on all the servers is acceptable. As clients leave the game, servers become underloaded and Matrix reacts by consolidating the load onto a smaller number of servers. For example, after 200 clients left the game (at time=75), server 3 became underloaded and reclaimed its "child" server (server 4). Matrix was similarly able to handle the subsequent appearance and disappearance of another hotspot (introduced at t=170) located at a different part of the map.

This result clearly demonstrates that Matrix, unlike static partitioning schemes, is able to deploy additional servers to react quickly and effectively to sudden load changes. This is significant, because game developers no longer have to a-priori over-provision their servers to prevent them from crashing (which would mar the game's reputation) under unexpected load spikes. These spikes could occur when particular areas in the game become popular suddenly, like the town hall during a town meeting, or by a massive influx of new game players entering the game (possibly due to an advertising campaign or a reference on Slashdot).

5.3.3 Matrix Overhead

In this section, we present microbenchmarks for Matrix's client switching time and bandwidth usage using the same scenarios and loads used in Section 5.2.

Time Taken to Switch Game Clients : Figure 11 shows the time required to switch clients between Matrix servers for different loads. This is the time from when a game server determines that a player needs to be switched to the point where it receives a reply from Matrix and can switch the player. This includes the time needed to transfer the player's state between servers. These results were obtained by running various loads for 5 minutes on a two server statically partitioned Matrix setup. The switching time is low ($<$ 0.03s), from the Figure, until the point where the game server becomes overloaded. Then the average and maximum switching times increase (the maximum can jump to $\approx$ 19s), along with the standard deviation. In practice, Matrix avoids these situations by adding new servers before the load becomes intolerable.

Network Traffic Sent Between Matrix Servers: Figure 12 shows the amount of overlap traffic sent between Matrix servers for different sized overlap regions. As expected, the overlap region size directly affected the bandwidth usage of each server. For example, for 80 clients, the overlap traffic ranged from 27 KB/s (100% overlap between the servers) to 0.9 KB/s

No. of Clients, Update Rate	Avg. (s)	Std.	Max. (s)	Min. (s)

RPG Scenario

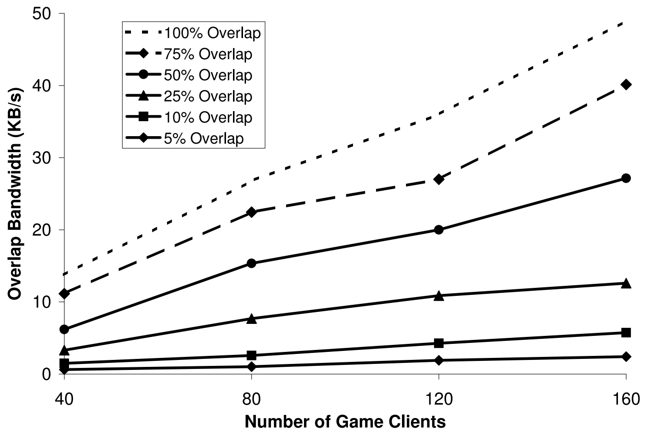
100 @ 1s	0.0024	0.0024	0.0144	0.0004
200 @ 1s	0.0077	0.0280	0.2752	0.0002
300 @ 1s	0.0123	0.0590	0.6955	0.0005
400 @ 1s	1.6284	4.0582	19.2887	0.0004

FPS Scenario

40 @ 100ms	0.0017	0.00329	0.0213	0.0003
80 @ 100ms	0.0042	0.00889	0.0735	0.0004
120 @ 100ms	0.0308	0.21181	1.9839	0.0003
160 @ 100ms	0.0471	0.34311	4.0482	0.0003

This Figure shows the time taken to switch clients (with different update rate) between Matrix servers under different player loads. The average time, standard deviation along with the maximum and minimum times taken to switch a client are presented.

Figure 11: Time to Switch Clients Between Matrix Servers



Each line denotes a different overlap region size (50% says that half the partition overlapped with another server). The clients all had update rates of 100ms.

Figure 12: Bandwidth Used by Overlap Traffic

(5% overlap). In contrast, the MC only exchanged $\approx$200 bytes with each Matrix server for every server split or reclaimed. These results suggest that for Matrix to scale, the overlap region must be small relative to the partition width.

5.4 Simple Asymptotic Analysis

The previous sections have shown a) that Matrix can satisfy actual game players, b) that it has excellent performance when hotspots occur (especially compared to static partitioning schemes), and c) that it has low overhead for a reasonable number of clients and servers. In this section, we show how Matrix can scale to a large number of clients. Since it is not possible to perform real world evaluations of Matrix's performance for our target environment of 1 million+ players and 1000 or more servers, we use analytical techniques to extend the detailed results from the small scale evaluations of individual games to predict Matrix's performance as the number of players increases. Even though such an analysis has limitations, it still provides useful insights into the scalability of Matrix. The analysis attempts to answer the following two questions: As the client population increases, how does the amount of consistency maintenance traffic sent between Matrix servers increase? Similarly, how does the amount of state traffic sent as clients switch between game servers increase? For Matrix to scale, both these two types of traffic should increase at most linearly with the number of clients in the system.

We analyze the asymptotic behavior of Matrix with a generic game, that uses a global map of WxL units with a visibility radius of R. We reuse our "split-to-left" strategy and split partitions along the L dimension, creating rectangular regions with constant width W. For this analysis, we assume that the user population is *uniformly distributed* over the entire map. Let the total number of users be U and the client density (computed as $\frac{U}{W*L}$) be ρ. Since the client distribution is uniform, Matrix will split to a uniform depth D, with 2^D equal-sized partitions, along all paths in the tree.

Let each server's high load threshold be T_h. The maximum partition length, l, that a server can have before splitting is given by $\rho * l * W \leq T_h$. The number of Matrix servers, N, required to support U clients is $\lceil \frac{U}{T_h} \rceil$, with the length of each partition given by $\lceil \frac{U}{T_h} \rceil$. Since a depth of D implies a partition length of $\frac{L}{2^D}$, to support a user population of density ρ, Matrix must split to a depth D_{req} given by $\lceil \log_2(\frac{U}{T_h}) \rceil$.

5.4.1 Consistency Maintenance Overhead

If each client generates B bytes/sec of traffic, the amount of client traffic per second that needs to be forwarded will be given by $B * \rho * min(2R, l) * W$ (as $\rho * min(2R, l) * W =$ number of clients in the overlap regions). Since this traffic must be forwarded to the corresponding consistency set (obtained from Principle 1, we can express an upper bound on total update traffic (in bytes/sec), both sent and received by a single server as:

$$Update = 2 * B * \rho * min(2R, l) * W * 2 * \lceil \frac{R}{l} \rceil^{1}, \quad (4)$$

5.4.2 Client Switching Overhead

An increase in the client population U can cause clients to switch more frequently between servers, with associated state transfer costs, especially as each partition gets smaller after multiple splits. To estimate this overhead, we assume that each client exhibits a modified fluid-flow movement model [26], with velocity v, and a direction uniformly distributed between $(0, 2\pi)$. The fluid flow analysis technique of Thomas [26] shows that the total rate of partition-crossings from a rectangular partition of width W and length l, via the two W edges, is actually independent of l and given by $\frac{\rho * v * 2 * W}{\pi}$. If we assume that each client switch requires a state transfer overhead of S bytes, the total (incoming and outgoing) switching overhead per second per server is given by:

$$Handoff = 2 * S * \frac{\rho * v * 2 * W}{\pi}. \quad (5)$$

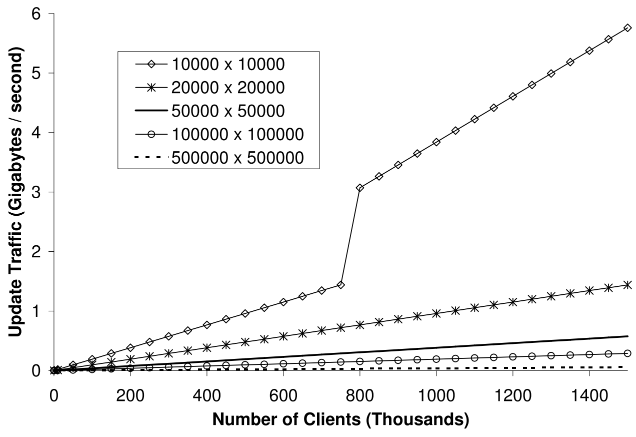
[1] Note that this equation assumes, pessimistically, that a Matrix server unicasts update packets individually to every member of the consistency set. In practice, the packet forwarding overhead on the network interface of the Matrix server could be lowered significantly by using network-layer multicasting protocols.

5.4.3 Results of Analysis

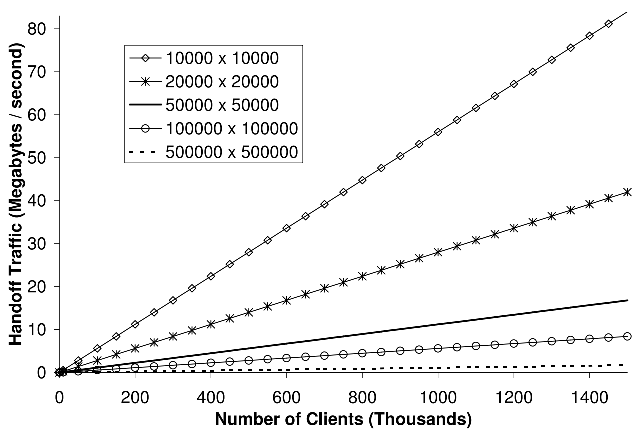
Equations 4 and 5 show that asymptotically, the amount of update and handoff traffic per Matrix server, similar to the number of deployed servers N, only increases linearly with the number of users. To understand the implications of these equations on our chosen 3 games, we use the measured runtime parameters of each of these games (shown in Figure 6 to compute the $Update$ and $Handoff$ overheads for different client population values U.

Figure 13 shows the computation results: as expected, the update overhead is the dominant factor in determining Matrix's scalability (as replicating updates is a much more frequent event than switching clients). Figure 13a, strongly suggests that to effectively scale to a large number of clients, we need to use a large map size. This is intuitive as the smaller the map size, the larger the client density which will result in a much larger number of clients being in overlap regions. At the largest map size with 1.5 million clients, the update rate is about 300 MB/s, while the smallest map size requires an exorbitant 6 GB/s. For this experiment, the amount of client traffic received by each game server was ≈3.6 GB/s (maximum 30000 clients each sending 12KB/s). Matrix thus significantly reduces, for a reasonably large map, the amount of client traffic that needs to be sent to other servers. These numbers also show that supporting a large number of clients, on today's machines, will not be possible without either a significant breakthrough in I/O bandwidth or by drastically reducing the number of clients on each server.

The figure also suggests that hotspots (smaller maps are equivalent to hotspots, both of which are characterized by higher client densities), can also be managed as long as Matrix doesn't split into inefficiently small partitions (such as maps smaller than 10000x10000). From Equation 4, other approaches to improving the scalability of Matrix would include reducing the packet send rate for game clients (although this may cause loss of synchronization between a game client and its server), transmitting smaller-sized packets per update (even though this might reduce the sophistication of the game), using a smaller radius of visibility (although this will lower the degree of interaction among players), and better use of network-layer multicast.



(a) Consistency Maintenance Overhead



(b) Client Switching Overhead

The graphs show the consistency and client switching overhead for different sized square maps (for simplicity). The high load threshold was set to 30000 players. To compute the consistency overhead graph, we used Quake 2's spatial packet size (60) and maximum packet send rate (300) as it had the highest update rate among the 3 games. The jump in the smallest sized map line occurs when the consistency set size changes because $l < R$. For the client switching overhead, we used the client movement velocity of Quake 2 (40) and the state transfer size of Daimonin (2198) to see what the worst possible switching overhead might be.

Figure 13: Overhead of Matrix

6 Related Work

There have been previous attempts at using scalable "grids" of servers to build distributed architectures for MMOGs [5, 20]. However, these solutions are still mostly in a formative stage. Peer-to-peer (p2p) architectures have also been proposed as a solution for MMOGs [12]. In these systems, players form localized groups and exchange messages directly with other players in the group, thereby allowing the system to scale. However, these mechanisms are unable to effectively handle hotspots and they do not clearly separate the game from the infrastructure, requiring each game to be intimately designed with the p2p network in mind. They also allow players to directly exchange game messages with one another, compounding the problems associated with collusion and cheating.

Commercial MMOG systems, such as Everquest [22] and Final Fantasy XI [23], carefully partition the game world between different servers to reduce the communication overhead between servers. To handle hotspots, they allocate multiple tightly-coupled (completely consistent) servers to handle the same partition, an approach that is neither efficient nor very scalable. Instead, Matrix techniques could be used by these systems, together with careful static partitioning, to efficiently and effectively handle hotspots and load fluctuations.

The notion of radius of visibility has been used extensively in the field of computer graphics where only objects in the immediate field of view are rendered. We apply this technique to the domain of multiplayer games. The use of localized consistency has also been used in previous systems to achieve lower latency updates at the expense of complete correctness. These include distributed shared memory systems [2, 13], databases [1, 6], and network protocols [10]. Unlike these previous systems, multiplayer games are nearly decomposable. This allows Matrix to use localized consistency to reduce latency without sacrificing any correctness.

Finally, there have been a number of algorithms to split virtual worlds among different servers. These include algorithms optimized for reducing inter-server communications [15, 18] and for preserving locality [8]. Our work complements these solutions. Matrix can use these algorithms to perform more optimal splits.

7 Conclusion and Future Work

In this paper, we have shown that it is possible to build, using localized consistency and on-demand mechanisms, an easy to use distributed middleware architecture that is able to satisfy the latency and scalability requirements of MMOGs. We have implemented Matrix and used its simple API to make three games (BzFlag, Quake2 and Daimonin) Matrix-compatible. The Matrix design is specially attractive because of its layered approach; by completely shielding the game from the actual mechanisms used to implement consistency, reliability and map partitioning, Matrix allows a game developer to use it with almost no modifications to the game client, and relatively simple modifications to the server code.

Experimental results show that Matrix outperforms static partitioning schemes when the workload exhibits unpredictable and dynamic skews. We also show, via a small user study, that Matrix, even with simple algorithms, is able to satisfy real users.

We plan to test the effectiveness of alternative splitting algorithms in the near future. In particular, we plan to investigate how easy it would be to use algorithms that keep track of where objects are in the game world and use that information when making splits. We also plan to investigate the effects of using heterogeneous servers (in terms of computational or networking capabilities) with Matrix. In particular, we plan to investigate what happens to Matrix if one or more servers starts slowing down while a game is being played (due to excess load or other factors), and if this situation can be resolved by dynamically adjusting the server thresholds. Finally, we plan to test Matrix with real MMOGs as soon as practicable.

References

[1] Adya, A. and Liskov, B. Lazy consistency using loosely synchronized clocks. *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC '97),* Santa Barbara, CA, Aug. 1997.

[2] Agarwal, A., Chaiken, D., Johnson, K., Kranz, D., Kubiatowicz, J., Kurihara, K., Lim, B.-H., Maa, G., and Nussbaum, D. The MIT alewife machine : A large-scale distributed-memory multiprocessor. *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic, 1991.

[3] Armitage, G. Lag over 150 milliseconds is unacceptable. `http://gja.space4me.com/things/quake3-latency-051701.html`, May 2001.

[4] Basch, J., Guibas, L. J., and Hershberger, J. Data structures for mobile data. *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 747–756, 1997.

[5] Bauer, D., Rooney, S., and Scotton, P. Network infrastructure for massively distributed games. *Proceedings of the 1st workshop on Network and System Support for Games (Netgames)*, pages 36–43, Bruanschweig, Germany, May 2002.

[6] Breitbart, Y., Komondoor, R., Rastogi, R., Seshadri, S., and Silberschatz, A. Update propagation protocols for replicated databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 28(2):97–108, 1999.

[7] Butterfly.net. *The Butterfly Grid*. `http://www.butterfly.net/`, Sept. 2000.

[8] Chen, J., Wu, B., Delap, M., Knutsson, B., u, H. L., and Amza, C. Locality aware dynamic load management for massively multiplayer games. *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of p arallel programming (PPoP)*, Chicago, IL, June 2005.

[9] DFC Intelligence. *Challenges and Opportunities in the Online Game Market - Executive Summary*. `http://www.dfcint.com/game_article/june03article.htm`, June 2003.

[10] Golding, R. A. A weak-consistency architecture for distributed information services. *Computing Systems*, 5(4):379–405, Fall 1992.

[11] Id Software. *Quake 2 Source Code*. `http://www.idsoftware.com/business/techdownloads/`, Apr. 2002. (Version 3.21).

[12] Knutsson, B., Lu, H., Xu, W., and Hopkins, B. Peer-to-peer support for massively multiplayer games. *Proceedings of the 23rd Conference of the IEEE Communications Society (Infocomm)*, Hong Kong, China, Mar. 2004.

[13] Lenoski, D., Laudon, J., Joe, T., Nakahira, D., Stevens, L., Gupta, A., and Hennessy, J. The DASH prototype: Implementation and performance. *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*, pages 92–103, Gold Coast, Australia, May 1992.

[14] Lety, E., Turletti, T., and Baccelli, F. Score: a scalable communication protocol for large-scale

virtual environments. *IEEE/ACM Transactions on Networking (TON)*, 12(2):247–260, 2004.

[15] Lui, J. C. S. and Chan, M. F. An efficient partitioning algorithm for distributed virtual environment systems. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):193–211, 2002.

[16] Malkin, G. S. Rip version 2, Nov. 1998.

[17] Moy, J. Ospf version 2, Apr. 1998.

[18] O'Connell, K., Dinneen, T., Collins, S., Tangney, B., ille Harris, N., and Cahill, V. Techniques for handling scale and distribution in virtual worlds. *Proceedings of the 7th ACM SIGOPS European Workshop*, Connemara, Ireland, Sept. 1996.

[19] Riker, T. Bzflag source code and online documentation. `http://www.bzflag.org/`, June 2003.

[20] Shaikh, A., Sahu, S., Rosu, M., Shea, M., and Saha, D. Implementation of a service platform for online games. *Proceedings of the 3rd workshop on Network and System Support for Games (Netgames)*, Portland, Oregon, Sep 2004.

[21] Simon, H. A. The architecture of complexity. *Proceedings of the American Philosophical Society*, 106:467–482, 1962.

[22] Sony Entertainment. *Everquest Live*. `http://eqlive.station.sony.com/`, Mar. 1999.

[23] Square Enix. *Final Fantasy XI Online*. `http://www.playonline.com/ff11us/index.shtml`, Oct. 2003.

[24] Square Enix. *Final Fantasy XI Online Press Release*. `http://www.playonline.com/ff11us/polnews/news1430.shtml`, Jan. 2004.

[25] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160. ACM Press, 2001.

[26] Thomas, R., Gilbert, H., and Maziotto, G. Influence of the movement of the mobile station on the performance of the radio cellular network. *Proceedings of the 3rd Nordic Seminar*, Copenhagen, Denmark, Sept. 1998.

[27] Toennies, M. Daimonin source code. `http://daimonin.sourceforge.net/`, Sept. 2003. (Version 0.96alpha1).

[28] Whitaker, A., Shaw, M., and Gribble, S. D. Scale and performance in the Denali isolation kernel. *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[29] Woodcock, B. S. Graphing the growth of mmogs. `http://pw1.netcom.com/~sirbruce/Subscriptions.html`, Mar. 2004.