Matrix: Adaptive Middleware for Distributed Multiplayer Games

Rajesh Krishna Balan[1], Maria Ebling[2], Paul Castro[2], and Archan Misra[2]

[1] Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA
[2] IBM Research Watson, 19 Skyline Drive, Hawthorne, NY 10532, USA

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, called Matrix, that is easily usable by game developers. We show experimentally that Matrix provides good performance, especially when hotspots occur.

1 Introduction

Online gaming is a rapidly growing market segment estimated to reach 100 million players and a USD $5 billion market value by 2008 [9]. A popular form of multiplayer gaming is the rapidly growing [24] class of massively multiplayer online games (MMOG) such as *Everquest* [19] and *Final Fantasy XI* [20], where hundreds or even thousands of players from across the world interact in a real-time 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, each server can handle at most 30,000 clients [7] whereas games like Final Fantasy XI claim to have at least one million registered players [21]. 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 instead, to 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 requires

G. Alonso (Ed.): Middleware 2005, LNCS 3790, pp. 390–400, 2005.

increasingly larger amounts of time as the amount of traffic and number of nodes in the system increases (due to increased player activity). However, 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* [18]. 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 behaviour 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. Another key Matrix design goal was ease of use. We achieved this by providing a clean and clear layering that hides the consistency maintainence details within an easy-to-use API (not shown due to space constraints). 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. Due to space constraints, we present a summary of these results.

In Section 2, we describe Matrix's design criteria while Section 3 presents the design and implementation of Matrix. Section 4 presents a summary of the evaluation while Section 5 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 their games 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 game developer preferred client-server architecture, as shown in Section 3, allowing the developer 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 would handle 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.

Support Multiple Gaming Platforms: Game developers frequently develop games for multiple gaming platforms; having to write new Matrix routines for each platform would hinder adoption. Our APIs do not require any new Matrix-specific routines for a new platform.

Simplicity: Building and debugging a large distributed system is a tricky endeavour. As such, Matrix intentionally uses the simplest possible algorithms and APIs. The simple algorithms allow Matrix to be easier to debug and maintain, and 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 a crucial factor 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).

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. Because 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 Matrix Design and Implementation

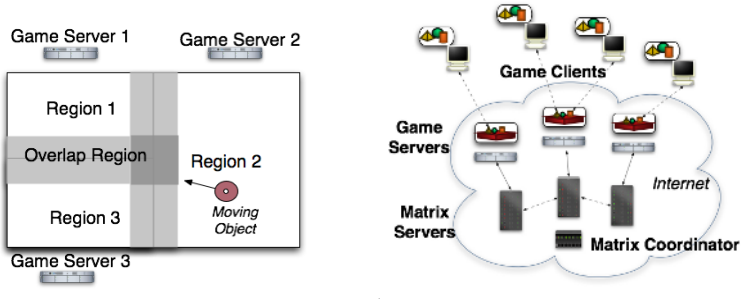
In this section, we describe Matrix's design and implementation, focusing primarily on the overall architecture and major technology components.

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).

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,



(a) Overlap Region between 3 Matrix Servers (b) Matrix Architecture

Fig. 1. Matrix Components

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 in σ. If $d(x,y)$ represents the distance-metric between points x and y,

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

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 utilize this sparseness by forming groups, called "overlap regions", of all points that have identical non-empty consistency sets (shown in Figure 1a).

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).

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 because 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).

3.2 Matrix Architecture

Figure 1b shows the Matrix architecture, that satisfies the design criteria in Section 2. A MMOG is deployed using Matrix with the MMOG developers providing game clients and game servers and the Matrix infrastructure providing 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* because 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 Matrix.

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 will split the game world between the overloaded server and a newly created game server and inform both the new and overloaded game servers of their new map ranges. The overloaded game server will then forward 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 will redirect 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 will decrease the load on the overloaded game server. However, if it is still overloaded, Matrix will split 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 will first check, using some non-Matrix external entity, for an available Matrix server. If a server is available, it will split its current map, keeping control of a subportion 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. Though simple, this algorithm still provides good performance as shown in Section 4.

The new Matrix server will then create a new game server and orchestrate the transfer of the global state, from the original (overloaded) game server to this newly-created game server. The overloaded game server will then switch game clients to this new server to ease its load. The amount of state associated with switching game clients is minimal (based on experience with the games used to test Matrix) and Matrix has efficient mechanisms (not described due to space constraints) to transfer this 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 will be the parent of the newly created Matrix server. When 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 will reclaim the partition and game state held by the child. All the game clients on the child's game server will be transfered to the parent's game server, after which the child Matrix server and game server will be removed from the game and returned to the resource pool. Matrix uses simple heuristics (not described) to prevent oscillations and ensure stability in the splitting / reclamation process.

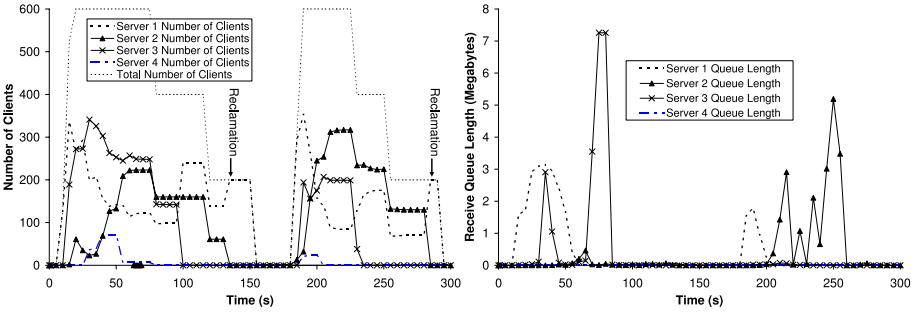
3.2.4 Matrix Coordinator (MC)

The MC creates the overlap tables used by Matrix servers to route spatially tagged packets. When a new Matrix server is used for the game, it informs the MC of the current map range and radius of visibility. 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 will then inform 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 [22]), 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 used in the latency-critical packet forwarding process (except for the rare non-proximal interactions). The MC can also be made reliable using well understood replication techniques.

4 Evaluation Highlights

Due to space constraints, we present just one detailed result showing that Matrix can handle dense hotspots automatically. The detailed evaluation results will appear in a longer version of this paper.



This Figure shows Matrix responding to a 600 client hotspot. 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).

Fig. 2. Hotspot caused by 600 clients

4.1 Behaviour Under Load and Hotspots

Matrix was designed to gracefully react to unexpected heavy loads and dense hotspots. We tested this by subjecting Matrix to loads far higher than what a static partitioning scheme could handle.

Figure 2 shows an experiment in which a hotspot of 600 clients (for a real shooting game called Bzflag [16]), far higher than a static partitioning could handle (results not shown), 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 2) 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 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, as 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 (E.g., due to an advertising campaign or a reference on Slashdot).

4.2 Summary of Other Results

In addition to Bzflag, we also tested Matrix with a role playing game called Daimonin [23] and a popular shooting game called Quake 2 [11]. For these three games, we showed that Matrix is able to outperform static partitioning schemes when unexpected loads or hotspots occur. In particular, Matrix is able to automatically use extra servers to handle the load while the static partitioning schemes just fail.

We also conducted microbenchmarks that showed that Matrix's overheads, in terms of switching latency and bandwidth usage, were acceptable. In particular, the overhead of using a central coordinator was negligible and the amount of traffic sent between Matrix servers corresponded directly to the size of the overlap regions.

We then conducted a simple user study, using Bzflag, that showed that Matrix is completely transparent to real game players. Even under heavy load, requiring Matrix to add servers, game players did not perceive any significant Matrix-induced performance degradation.

Finally, we performed a simplistic asymptotic analysis of Matrix. This analysis reaffirmed the microbenchmarks and suggested that a) Matrix can scale to a large player population ($>$ 1,000,000 players and 10,000 servers) only if the number of players in the overlap regions is small relative to the total number of game players, and b) that Matrix scalability is ultimately limited by the maximum I/O capacity of individual servers.

5 Related Work

There have been previous attempts at using scalable "grids" of servers to build a distributed architectures for MMOGs [5,17]. 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 [19] and Final Fantasy XI [20], 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 can 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. However, we are applying 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]. However, 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 [14,15] and for preserving locality [8]. Our work complements these solutions and Matrix can use these algorithms to perform more optimal splits.

6 Conclusion

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 allow three games (BzFlag, Quake2 and Daimonin) to use Matrix. 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.

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., Lu, H., and Amza, C. Locality aware dynamic load management for massively multiplayer games. *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel 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.
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. 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.
15. O'Connell, K., Dinneen, T., Collins, S., Tangney, B., 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.
16. Riker, T. Bzflag source code and online documentation. http://www.bzflag.org/, June 2003.
17. 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.
18. Simon, H. A. The architecture of complexity. *Proceedings of the American Philosophical Society*, 106:467–482, 1962.
19. Sony Entertainment. *Everquest Live*. http://eqlive.station.sony.com/, Mar. 1999.
20. Square Enix. *Final Fantasy XI Online*. http://www.playonline.com/ff11us/index.shtml, Oct. 2003.
21. Square Enix. *Final Fantasy XI Online Press Release*. http://www.playonline.com/ff11us/polnews/news1430.shtml, Jan. 2004.
22. 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.
23. Toennies, M. Daimonin source code. http://daimonin.sourceforge.net/, Sept. 2003. (Version 0.96alpha1).
24. Woodcock, B. S. Graphing the growth of mmogs. http://pw1.netcom.com/~sirbruce/Subscriptions.html, Mar. 2004.