Running Quake II on a grid

G. Deen
M. Hammer
J. Bethencourt
I. Eiron
J. Thomas
J. H. Kaufman

As a genre of computer games, the massively multiplayer online game (MMOG) has the promise of enabling up to tens—or even hundreds—of thousands of simultaneous players. This paper describes how we began with an existing single-server online game engine and enhanced it to become a multiserver MMOG engine running on a grid. Other approaches require that a game be specifically designed to scale to MMOG player levels. Our approach, using IBM OptimalGrid middleware (which provides an abstracted underlying grid infrastructure to an application) allowed us to reuse an existing game engine without the need to make any significant changes to it. In this paper we examine the design elements needed by an MMOG and provide a practical implementation example—the extension of the id Software Quake II® game engine using OptimalGrid middleware. A key feature of this work is the ability to programmatically partition a game world onto a dynamically chosen and sized set of servers, each serving one or more regions of the map, followed by the reintegration of the distributed game world into a seamless presentation for game clients. We explore novel features developed in this work and present results of our initial performance validation experiments with the resulting system.

INTRODUCTION

The computer game industry is deploying a new genre known as the *massively multiplayer online game* (MMOG), characterized by large numbers of clients—ranging from several hundred to hundreds of thousands—playing simultaneously. Until now, the creation of such a scalable game has been done primarily by custom-designing game engines unique to the specific MMOG. Efforts to create reusable MMOG game development and game engines have been underway, Butterfly.net[1] being an example of a commercial game development environment for MMOGs. In its work, Butterfly.net obtained patents related to the development of games and games using grids. Subsequently, it has updated its business plan and renamed itself Emergent Game Technologies. At the time of the writing of this paper, details about this new entity's game architecture were not available. The issues surrounding the building of MMOGs and large-scale virtual environments[2] have become an area of interest and study, with academia publishing studies on general architecture, intelligent on demand provisioning,[3] server selection,[4] distributed communications mod-

els,[5] distributed server location schemes,[6] game artificial intelligence, and agent-based testing frameworks for game developers. Additionally, the application of MMOG technology to building large-scale agent-based simulations[7] for research on non-game-related commercial customer relationship management systems is also underway. The complexities of large-scale group interactions[8] within online games, that is, exploring the communities that come about within the virtual worlds of the games, has also become a topic of research.

Many techniques have been developed to distribute both massively parallel applications and connected parallel applications, such as cellular automata and finite element modeling.[9] MMOGs—like other applications that stress the resource limitations of a single system—are excellent candidates to which the scalable computing power of grids can be applied, such as the approach taken by Emergent Technologies[1] and that presented in this paper.

MMOGs are especially demanding in their low communication-latency requirements. In computing, as in other systems, increasing the number of parts, in this case the number of computers, increases the likelihood of failure of a part. To provide a stable and long-lasting game for players, MMOGs running on a grid require fault tolerance and dynamic load balancing, just like scientific grid applications. However, unlike scientific applications that tolerate interruption for checkpoint and logging, online games do not permit interruption or latency degradation during load balancing or system maintenance.

Because considerable performance optimization and game-play balancing have been done to single-server multiplayer engines such as id Software Quake** and Valve Half-Life**,[10,11] it would be desirable if the effort put into such engines could be reused to run MMOGs. Instead of designing new engines specifically for MMOGs, the reuse of existing optimized engines would allow many of today's games to be scaled up to MMOG level and allow game developers to continue using game engines with which they are familiar. Many popular engines apply similar architectures, offering the hope that techniques for scaling up of game engines to MMOG levels for one engine can be reapplied to other engines. Ideally, such scaling techniques would be done in a generalized manner, rather than specifically enhancing any one particular engine, helping to broaden the applicability of the techniques.

In this paper we report the results and lessons learned in enhancing an existing single-server multiplayer game engine to scale to the MMOG level. We chose the popular open-source Quake II**[12] game engine from id Software for this work, but have attempted to develop reusable generalized techniques in performing the enhancement. This enhancement was accomplished by using the OptimalGrid[13] autonomic grid middleware from IBM Research. This middleware was originally designed with scientific and engineering applications in mind; however, the OptimalGrid object model and dynamic load-balancing features were easily adapted to asynchronous online games.

In this paper, we examine the key issues in extending a game engine to running in a multiserver environment; we introduce the Quake II game engine and OptimalGrid middleware. We then provide an overview of the resulting system, outlining its major components, followed by an exploration of the details of the design. We introduce the runtime environment developed to manage the execution of the MMOG game engine and discuss the additional debugging challenges inherent in a large distributed system, along with the solutions we used. We then present the results of the performance experiments we performed on the system, and lastly, our conclusions and a brief outline of future work.

MASSIVELY MULTIPLAYER ONLINE GAMES

Many types of parallel computational problems have been considered in research and implemented in mature systems. They range from problems with moderate connectivity, such as the simulation of finite element models, to massively parallel tasks, such as the search for optimal Golomb rulers.[14,15] MMOGs however, present a set of unique constraints. Like many scientific simulations, the game worlds found in MMOGs are often well-suited to division into spatial regions that may be run in parallel, albeit connected regions.

Game-world partitioning

Game worlds typically are continuous like the real world. This means that each spatial division has strong connectivity to the regions that neighbor

(touch) it; actions and events in one region affect objects in neighboring regions. For example, a gun fired in one region of the world may cause damage to a player in another region that lies along a line-of-sight path from the first. Long-range interactions between regions can exist, meaning in the example that there may be multiple regions along the line-of-sight path, with each region possibly being hosted on a different server. Thus connectivity between regions is not limited to those that are "physically" adjacent, but instead, connectivity exists between any region and all the regions that are visible to it. For example, in long hallways and large open areas, a player's local actions must be observable to players in any other visible region, regardless of which server is controlling that region. Any implementation of spatial partitioning of the game world into discrete regions assigned to different servers must provide a mechanism to propagate events to servers holding adjacent regions, and it must also have a mechanism to propagate such events along the full path of connected regions, and events must be propagated in a timely manner so that no cause-and-effect delay is introduced that exceeds that intended by the game designer or would be experienced if the game regions were all hosted on a single server.

The dependencies between spatial regions of the world are mitigated somewhat by the presence of barriers such as walls, but some long-distance connections typically remain. This situation complicates the problem of dividing the world into regions. How can a parallel system take advantage of the near-neighbor connectivity of regions with multiple obstacles while still allowing distantly connected regions to correctly influence one another? An intelligent algorithm for positioning region boundaries should take into account the underlying mechanics of the game that may or may not allow one point to influence another.

Data exchange

One way to help address the problems of strong connectivity is to introduce a global, shared message space, or *whiteboard*, for communication between portions of the parallel system. In the most extreme case, if every one of n servers was responsible for a region of the world connected to some region on every other server, connecting the servers directly to one another would result in $O(n^2)$ connections across the network, or $n-1$ per server. Introducing a single whiteboard as an intermediary, however, reduces the total connections to $O(n)$, or one per server. Of course, this improvement comes at the cost of increasing latency in sending messages and limiting the collective bandwidth of all communication channels in the system to that of the whiteboard. A more scalable approach is to use a number of whiteboards, each responsible for forwarding messages between a set of servers that are nearby (in the sense of game-world regions). Each server then needs to connect only to several whiteboards.

Partition granularity

A key problem involved in the partitioning of the game world is choosing the degree to which the world is subdivided. For a one-time or static partitioning of the game world, a significant trade-off must be made. If too few regions are used, then fewer servers may be used, but regions can become overloaded with players and other active entities. If too many regions are chosen, the area of the game world covered by a region becomes quite small. While this allows for more servers to be used, it can result in excessive data exchange among servers due to the number of regions connected to each region; player movement between servers will be very frequent, and other game entities, such as weapons fire, will be required to cross numerous servers. A practical strategy for choosing a partitioning granularity takes into consideration the maximum number of servers upon which the game will be hosted, the complexity of the interconnectedness of regions, and features specific to the game that affect event propagation and generation.

■ A key problem involved in the partitioning of the game world is choosing the degree to which the world is subdivided. ■

Although some effort at selecting an optimal allocation of servers and assignment of regions can be made before launching the game, such choices become outdated and invalid quickly as the result of game play. Thus, to achieve and maintain an optimal use of servers requires adaptive reassignment of regions and reallocation of servers (adding and removing) during game play. This is because

server load arises primarily from the players active on a server and the players moving within the game world—and thus moving between servers. These activities can continuously change the optimal assignment of regions and use of servers.

Latency

Apart from the issues arising from the topology of the game world, MMOGs have another set of distinct challenges. These derive from the interactive nature of games. One would like the solution of a scientific problem to be completed as soon as possible, but the precise time taken by each successive step is not typically a concern. Games, however, have stringent constraints on the time taken to compute successive states of the world and the latency in client/server communications. This is especially true because it is normally necessary to perform all game "physics" (evaluation of the rules of the game world) on servers to prevent players from cheating by altering their client software. Consider the sequence of events triggered in a typical game when a player moves forward. After the user chooses to move and presses the corresponding key, the command is sent to the server, which computes the movement, and the new position of the player is then sent back to the client. Players do not get any visual feedback that they have moved until this round-trip communication has taken place. Players are sensitive to delays in this feedback loop as small as tens of milliseconds. To address this sensitivity during periods of unusually high latency, the Quake II client attempts to predict the server response to provide feedback to the user sooner. However, this prediction is error-prone and considered a last resort.

These latency requirements significantly constrain parallel architectures for MMOGs. If a distributed architecture requires switching of network connections between clients and servers or transmission of game objects such as players or projectiles between servers, those transitions must take place in a time frame sufficiently short to allow a seamless user experience.

One strategy to reduce critical latencies is to treat aspects of the system as asynchronous. Rather than having every server act in lockstep, exchanging messages at regular intervals, as is essential in scientific simulations, it can be advantageous to allow different components of the system to run freely, sending each other events as soon as the need arises. In a system for running MMOGs on multiple servers, this may mean allowing the servers to generate states of their portion of the world out of sync with each other and to send each other messages as soon as it is useful to do so.

QUAKE II

Quake II is a popular multiplayer (but not massively multiplayer) game developed by id Software. Released in late 1997, it is representative of the genre of games known as *first person shooters*—games in which the player is presented with a three-dimensional (3D) view rendered from the perspective of the game character and whose action is centered on the player firing weapons at adversaries. Like most other first person shooters (FPSes), multiplayer games in Quake II take the form of *death matches*, in which players run around a level composed of rooms, tunnels, and outdoor areas attempting to shoot and kill their opponents. Although almost seven years old, Quake II is particularly representative of many online games. Its engine has been used in a number of other well-known games including Valve Half-Life**[10,11], Xatrix Kingpin: Life of Crime**,[16] and Soldier of Fortune**.[17] Also, its engine is architecturally similar to that of the more recent Quake III**.[12] More important, id Software released the complete source code under the GNU General Public License (GPL) in 2001, allowing us to modify it for research purposes. The engine is written in a combination of highly optimized C and x86 assembly.

Multiplayer games in Quake II are run by a single server. All network communications are over User Datagram Protocol (UDP). The designers of the Quake II network protocol chose to use UDP to reduce latency and also because it is not useful to retransmit some time-sensitive messages. For example, suppose a Quake II server sends a client an update on the state of the world so the client may render it, but the packets are lost. By the time the client could discover that the packets were dropped and request that the server retransmit, the server would have likely generated the next state of the world. Rather than retransmitting, the server should send the most recent information.

All game logic and physics are carried out on the server; the client is essentially a graphics-rendering and client-input engine. The client continuously

sends packets to the server with the state of the user keys and mouse position. The server keeps all game-state information and sends clients updated positions and appearances of entities in the game world 10 times per second, or every 100 ms. To make motion appear smooth, the client interpolates between the updates in the 10 Hz stream. In moments of unusually high latency, the client also attempts to predict the contents of the next sever update, but this is error-prone and considered a last resort.

We now give a few more details of the production and structure of Quake II levels, which will be useful later. A number of levels were developed by id Software and shipped with the game, and many more have been created by players. Levels (self-contained scenarios within the game) can be downloaded from the Internet and loaded into the game at runtime. After the geometry of a level has been created using specialized map creation tools, the positions and orientations of walls and other surfaces in the level are used to create a binary space partitioning (BSP) tree. The BSP tree divides the entire level into a number of small, irregular (usually), convex polyhedrons, each of which corresponds to a leaf in the tree. These BSP leaves are further organized into *leaf collections* composed of several adjacent leaves. A typical leaf collection may be the size of a portion of a room. After the leaf collections have been generated, each pair of leaf collections is checked for the presence of line-of-sight visibility between them, and this relation is stored in the level with the leaf collections.

OPTIMALGRID

IBM OptimalGrid technology is research middleware designed to hide the complexity of creating, managing, and running, large-scale parallel applications on any kind of heterogeneous computational grid. OptimalGrid was created, in particular, to address scientific and technical computing problems that are parallel and connected, that is to say, not massively parallel. However, the OptimalGrid object model is general enough to handle a wide variety of other coupled parallel applications, including MMOGs and massively multiplayer online role-playing games (MMORPGs). OptimalGrid automates the task of resource allocation on a computational utility or grid to optimize the performance of running applications. It efficiently and automatically partitions a given problem throughout a large collection of computer resources and manages communication between the nodes. It also adjusts to the dynamic grid environment by providing autonomic functionality in the middleware layer to re-optimize the complexity of running problem pieces to match the changing landscape of resource capabilities on a grid.

OptimalGrid's runtime model uses three different services that are placed on computers in the grid: autonomic program manager (APM), computer agents (CAs), and TSpaces servers. The APM oversees the execution of the application and contains any load-balancing and global-scheduling control used by an application. The APM directs the work assigned to and executed by each of the CAs. CAs hold and execute the work units assigned to them by the APM. Each CA has a variable problem partition (VPP), which is the set of work units assigned to the CA. The size of this set can be varied by the APM—hence its name.

Communication between grid nodes is accomplished by exchanging messages with a set of one or more distributed whiteboards. This distributed Linda[18,19] model for communication is embodied in IBM TSpaces[20] technology. Communication by reading messages from or writing messages to a whiteboard server has some additional overhead, but offers several important advantages:

1. The whiteboard is both a communication system and an in-memory database. This greatly simplifies the implementation of both load balancing and fault tolerances. It also allows caching, staging, and checkpointing of persistent data (that may reside in a relational database).
2. In a highly connected problem, a whiteboard model can actually reduce the load. If each problem piece must exchange data with N neighbors, a peer-to-peer architecture requires $N-1$ connections per node. In a distributed whiteboard architecture, each node reads from one whiteboard and writes to n whiteboards, where $n << N$.
3. When the communication load is high, OptimalGrid takes advantage of the TSpaces multiread and multiwrite operations that group messages, thus lowering overall message traffic. So long as the computational load is correctly balanced with the communication cost, this can actually lower the application back-end latency. When the communication load is very low, OptimalGrid

takes advantage of a wait-to-read or wait-to-take mode. Thus nodes can register for callbacks and receive data only when it is available, reducing network load by eliminating query overhead.

OptimalGrid does not preclude the use of other peer-to-peer communication modes, and one can even locate a whiteboard on each grid node and perform all reads from main memory. However, we found that even for an FPS application like Quake II, communication through a set of distributed whiteboards delivers adequate or even excellent performance with good scalability.

The OptimalGrid object model assumes that an application can be described as a graph where the nodes on the graph contain data, methods, and pointers to neighbors. In OptimalGrid terminology, these nodes are called *original problem cells* (OPCs). OPCs are the "atomic" problem units or the smallest pieces of a problem that represent a unit of computation. In general, OPCs interact with their neighbors, sharing information to produce a larger, big-picture computation. Therefore, an OPC must communicate its state with its neighboring OPCs. During load time, OPCs are grouped into collections. These collections are precomputed and have predefined dependencies on other collections. Each computational node is assigned one or more OPC collections. This defines both the computational workload and the communication workload (the *collection edges*) for that node. The partitioning of the entire initial set of OPCs into collections and the assignment of OPC collections to actual computers on the grid must minimize the overall processing time. The VPP for each CA is assigned zero or more OPC collections, the actual number of which is changeable through a load-balancing operation by the APM. Thus, a computer runs a CA that holds a VPP, which in turn holds zero or more OPC collections that are each made up of one or more OPCs.

Processing time is composed of three components: computation time, communication time, and, in the case of synchronous applications, idle time. To efficiently parallelize the application, the ratio of computation to communication for each node must be minimized.

The computation time per server is primarily a function of the number of players within the server OPCs. When the OPCs are initially partitioned into collections, however, we cannot know precisely where players will be positioned; therefore, we attempt to balance the area of the game world within OPCs, OPC collections, and servers as an approximation.

Additionally, when distributing OPC collections among servers, we wish to give each server a set of adjacent collections in order to minimize transfers of players and other game entities between servers.

Given these considerations, the BSP-tree leaf collections of a Quake II level are a natural choice for the OPCs. To group the OPCs into collections, we place them into an octree. (An octree is a tree data structure that organizes 3-dimensional space. Each node represents a cuboid volume and has eight children.) The nodes of the octree are recursively subdivided until the number of nodes reaches the desired number of OPC collections. Each time it is necessary to divide a node, the largest node is selected. The resulting collections are then approximately equal in size and consist of spatially adjacent OPCs. An OPC collection is considered to be dependent on another collection if there is line-of-sight visibility between an OPC in the first collection and an OPC in the second collection. This is determined by checking for visibility between the corresponding leaf collections, as stored with the level by the Quake II map-creation tools. This method was tuned to ensure that the resulting OPC collections were small enough that a server could handle at least several (even at their busiest in terms of player presence and activity). Larger OPC collections may result in the inability to effectively balance the load among servers. As a side note, for a game to be effectively parallelized and run on grid middleware, it must provide some hints about the connectivity among locations in the game world. Quake II does this by providing the line-of-sight visibility information between its leaf clusters within the map file.

At runtime the grid nodes report their real-time performance data. This data is collected by the OptimalGrid load balancer, which watches over the running parallel application and attempts to balance the load across the computational nodes. This task is accomplished by reassigning OPC collections from loaded to less busy machines so as to harmonize the workload across the servers being used by the game.

The actual OPC collection to be moved should be chosen with great care. The estimated communication overhead should be low relative to the expected gain in computation time so that the overall performance gain will be beneficial. Additionally, if all the available servers are being used to the limit of their reasonable capacity, the system can add new servers to meet demand. Likewise, when the servers supporting the game world are underutilized, the world can be consolidated onto fewer servers, freeing those that are no longer needed to meet demand.

In *Figure 1A* we show a fragment of a BSP tree. The root node of the tree represents the first partitioning of the world space of the game into two halves, separated by a plane. Each internal node represents one half the space represented by its parent node and also splits this space into two halves for its children. The leaves of the tree do not introduce any additional splitting of the space. In Quake II, the BSP leaf nodes can represent a relatively small space compared with the size of typical game objects (such as players), so they are quite numerous. Therefore, the game defines leaf clusters to be groupings of leaf nodes that share common attributes, such as visibility information. The game map format contains this BSP tree along with information about the relevant relationships among leaf clusters. In our system, we create a one-to-one mapping between leaf clusters and OPCs. When we form OPC collections, we use the information the map file provides us about leaf clusters to guide us in creating edges between OPC collections. An example fragment of the graph of connected OPC collections is shown in *Figure 1B*. In practice the Quake II map file is generous in associating leaf clusters for visibility, and as a result, the graphs of connected OPC collections approach completeness (a graph is complete when every pair of nodes is connected). Because this is undesirable for distribution, we employ stricter heuristics to determine which OPC collections should be connected.

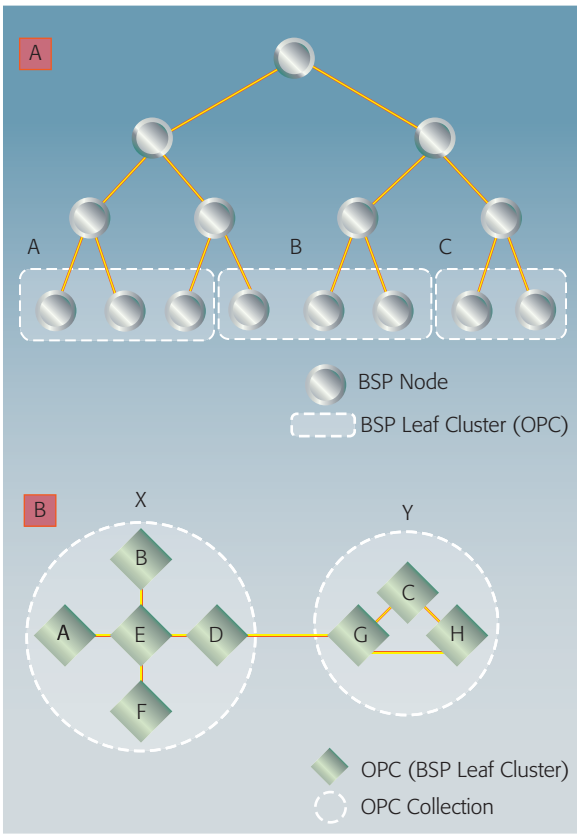

Figure 1
Mapping: (A) Binary space partition (BSP) tree to IBM OptimalGrid original problem cells (OPCs); (B) BSP leaf clusters to OptimalGrid OPC collections.

SYSTEM OVERVIEW

Figure 2 illustrates the core components that make up OptimalGrid and the components that comprise the Quake II game environment running on OptimalGrid. The following describes each of these components and conceptual elements and its role in the overall system.

- *Autonomic program manager*—A service responsible for coordinating the start up and operation of a distributed program. In this case, the APM coordinates the distributed game and can be used to manage its computing resources and dynamically adjust allocations of these resources during the game.
- *Back-end boundary*—Defines the boundary between the infrastructure of the grid and the outside world. Things on the inner side of this boundary are assumed to be owned and operated by the party hosting the distributed game; clients outside of this boundary could potentially be anyone with access to the system proxies and a copy of the game client.
- *Bot (short for robot)*—A modified game client that is given enough artificial intelligence to wander about the game world, stressing the system infrastructure in the process. Bots are meant to simulate clients when enough players are not

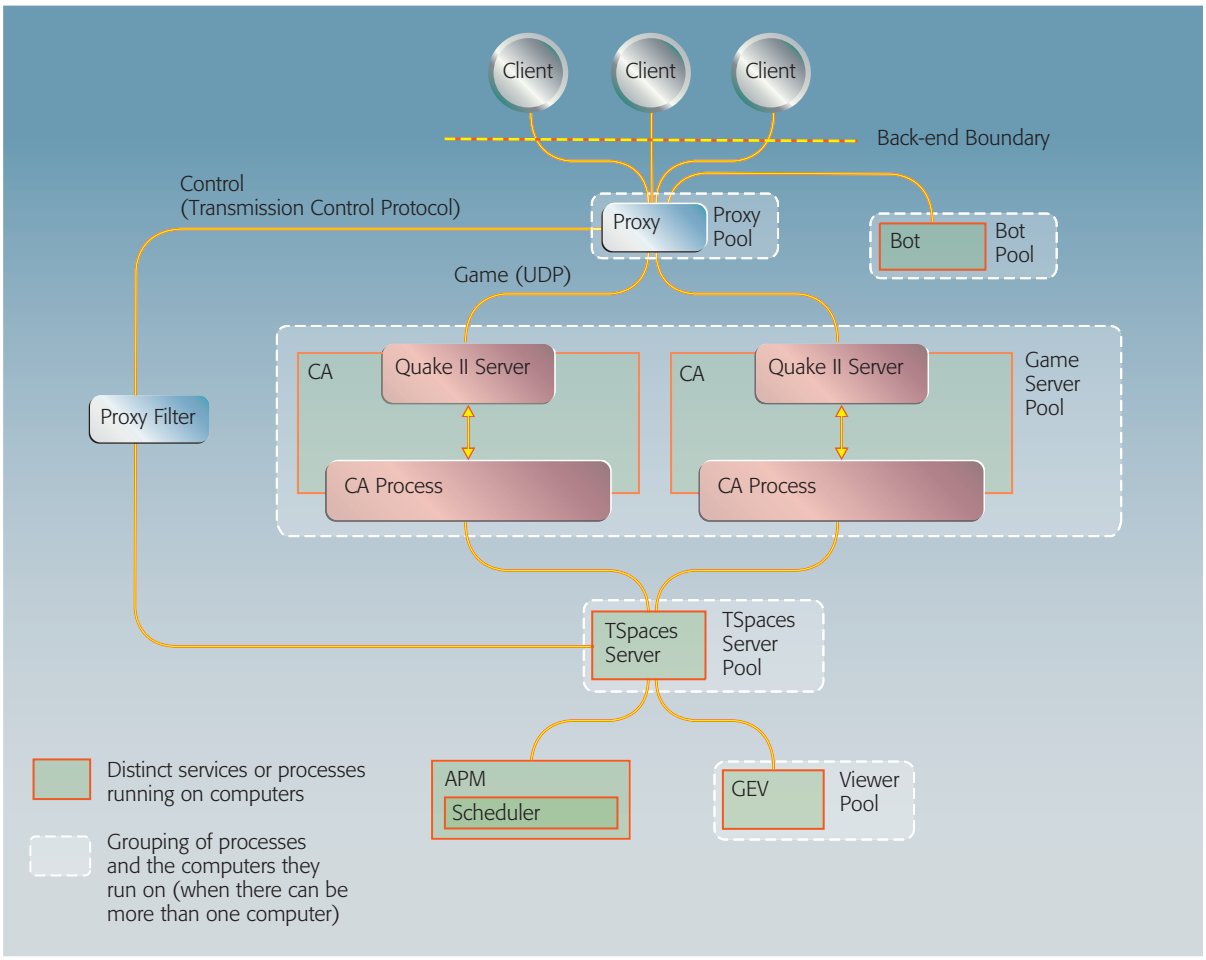

Figure 2
System components and communication paths

available or when there is a need to control participation in the game for testing and debugging purposes.

- *Client*—A human user not necessarily associated with the hosts of the distributed game. The clients represented in Figure 2 can be game clients in the sense of software connecting to the proxies via a network connection and human users interacting with the system . . . and ultimately the virtual game world.
- *Compute agent (CA)*—OptimalGrid terminology for a network node with a piece of the grid application problem. In this case, the problem is hosting the game world, and having a piece of that problem means being responsible for a region of that virtual world and everything that moves through it. The CA is a component made of both general OptimalGrid logic and application-specific logic. In our case, this is logic that pertains to hosting a Quake II server instance.
- *CA process*—A Sun Java** process that extends and ultimately uses the OptimalGrid code base to run application-specific logic. In our case, this is logic to interface the CA to an associated Quake II server instance.
- *Game server*—A term we use to refer to a CA or CA process given the task of running an instance of the Quake II server. We usually talk about game servers in a context in which we need to refer to them collectively, or in the context of other components of our system.
- *Console*—A visual tool used by a human to see and manipulate the current state of the game on the grid. (It is labeled on Figure 2 as *grid-eye view* [GEV].) Using this tool, a user can see which CA nodes are currently hosting pieces of the game

world, how those pieces are arranged, and where players and projectiles are in respect to them. A user at the console also has the option of moving game-world pieces among the game server pool to demonstrate the overall ability of the system to handle the world in a dynamic way.

- *Proxy*—A lightweight networking daemon written in the C language to perform various tasks relating to interfacing a pool of game clients with a pool of potential servers. Among other things, it filters and routes the game traffic between these two pools by using information about the current distribution of the game world and the current set of game servers in the game server pool.
- *Proxy filter*—Interfaces the proxy (written in the C language) to a Java-based TSpaces communication system. Using this interface, the proxy gets initialization information about the game world as well as updates about changes to the server and proxy pools.
- *Quake II server*—An instance of a Quake II game server modified to work under the control of OptimalGrid.
- *Scheduler*—A pluggable unit that can be either general or application-specific, which gives the APM logic the ability to balance the load on the grid that it is managing.
- *TSpaces server*—An implementation of the Linda model for representing persistent shared state among distributed nodes through spaces holding tuples. These tuples collectively hold all the information necessary to coordinate the operation of the system and keep its current state.
- *Tuple space*—A whiteboard space on a TSpaces server for holding tuples. The Linda model defines operations for reading and writing tuples in these spaces, as well as making queries for tuples meeting some match criteria.

DESIGN ASSUMPTIONS AND DECISIONS

We started with a small set of assumptions about how the system would be composed and what it ought to be capable of when we were finished. The following are our initial assumptions:

1. The use of Quake II as our proof-of-concept game server and client was chosen because it was available under the GNU GPL; it was simple enough to rework in a short period of time; and it was fast-paced enough to make perceivable lags in our underlying infrastructure detectable, which was important to demonstrate that our solution was capable of meeting the performance needs of games.
2. The use of OptimalGrid as our grid middleware technology was chosen because of its inherent ability to scale systems across a grid and provide the needed communications and load-balancing technology to make the game world both distributed and dynamic.
3. The use of TSpaces was chosen because it was already used by the OptimalGrid system for communication, and it provided a scalable communication system for managing the game-system global state across all the individual nodes of the grid.

   We would make only the most minimal modifications to the Quake II server and client. Our intention was to demonstrate the application of grid technology to online games, not to create a new and improved Quake II engine.

By deciding to grid-enable an online game, we committed to having many game servers, many proxies, and many TSpaces servers.

■ We would make only the most minimal modifications to the Quake II server and client. ■

Server boundaries

One problem with splitting a once single-server game into distributed pieces is that the game space now has boundaries and, as objects like players and projectiles move through game space, they need to be able to see across and move across *server boundaries*. A popular solution to this problem is to make each section of the game world cleanly divisible from the rest and small enough to exist on a single server. This means, however, that there is no way for the player to see what is happening on a server without actually being in the region of the game world on the server. It also means using a contrived map that just connects otherwise disjoint game worlds through some kind of game element like a door or teleportation device. Because the map is contrived so that the state of each server is self-contained, there is no communication issue among the pools of game servers and game clients. Just as with a single game server for a set of clients, the

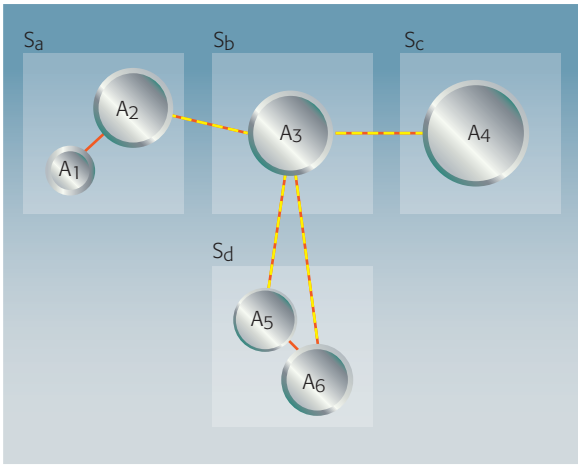

Figure 3
Game map partitioned into six areas ($A_{2–6}$) and distributed among four game servers, $S_{a–d}$

system has a one-to-many mapping between game servers and game clients.

We did not want to create or use contrived maps, nor did we want just to devise a system that had no state dependencies among the system game servers. To have a satisfying and general solution required the division of a premade world into a form that could be managed by many servers.

Following the OptimalGrid approach, we created a *problem builder* to make this splitting a start-time process that happens only when the system is initialized. Thus, before the game world is adopted by the game servers, it is split into indivisible units that we call areas. These *areas*, as they are implemented in our system, correspond to the OptimalGrid concept of an *OPC collection*. One or more areas are then allocated to each of the active game servers. Once defined, these areas do not change shape or size, but the assignment of areas to game servers can change. As load-balancing occurs, areas are traded among game servers, and the collections are thus redefined. Any two areas that are adjacent in the original game world but are grouped apart on separate servers constitute a server boundary. The problem builder and the dynamic load-balancing algorithms can identify which areas are mutually visible and, whenever possible, server boundaries are chosen to minimize both the surface area and visible interfaces between the servers. There are, however, always some visible connections. This boundary is literally a surface in the map (usually a plane) that separates area volumes managed by separate servers. When a game object, such as a player or projectile, crosses such a boundary that separates server S_a from server S_b, S_a must remove the game object from its state and communicate its arrival and relevant associated state to S_b.

In ***Figure 3*** we show a simple game world of six areas managed by four game servers. Solid lines illustrate areas grouped on the same server that are mutually visible and connected, and dotted lines show connections to areas placed on other servers. We needed to address the problems associated with allowing paths of visibility and influence to cross such server boundaries. Say, for example, that a player in area A_2 is facing into A_3, and something in A_3 is moving or otherwise changing state. S_b needs a way of keeping players up to date who are currently in S_a areas.

Proxy

In principle, game servers and game clients could talk to each other directly, as they do in the original Quake II game design, but this would introduce some drawbacks. In the original design, there is exactly one server managing the game world of one or more clients. In our system, we assume there can be both many clients and many servers. Having clients and servers talk directly would require connection-switching logic in the game client, which had already been written under the assumption that there was only a single game server.

Additionally, forcing direct contact between clients and servers means potentially exposing much of the internal infrastructure. This certainly includes game servers, but it also includes other services not used directly by the game clients, that is, services used by the grid infrastructure as it communicates internally and manages its subsystems. Exposing these systems means giving them addresses that are accessible from all potential game clients, which may include the entire Internet. In some cases this may be acceptable, but it would also be desirable to completely hide these machines from the end user both for security and for added indirection and abstraction of the grid system working behind the scenes. We thus implemented an intermediary proxy to manage the many-to-many connectivity that is required when more than one server is used.

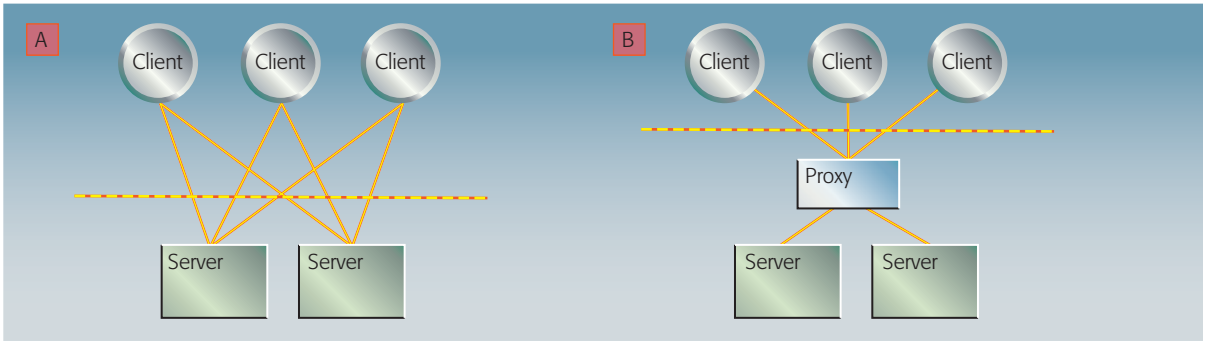

Figure 4
Back end connections: (A) When clients and servers are connected directly, the back end is exposed, and the number of required connections is large; (B) when a proxy (or set of proxies) couple clients and servers, the back end can remain private, and the proxies can multiplex connections between clients and servers.

This approach, and the alternative direct connection approach, are illustrated in ***Figure 4***.

In Quake II, the game server generates a new world state every 0.1 second. At this same rate, the server must send this state to all of the clients that need it. In our system, this includes both clients that are in the region of responsibility of this server and clients that can "see" into this region. Without a proxy, it is difficult for the server to get this information to the right clients without having to keep a lot of repetitive information. For example, it must know about clients that it is not managing directly so that it can send updates to them. As an alternative, the server could send its state to neighboring servers, which could then forward it to the clients that they are managing, but this approach again requires servers to be responsible for keeping other servers informed about their continually changing state.

In our system, we eliminate this complication by placing a pool of proxies between the clients and servers. When a client wants to play the game, it selects a proxy, connects to it, and stays connected to it even as the player traverses the game world. The proxy performs the act of connecting to the game server responsible for the region of the world in which the player is located. As the player moves through the world, and hence into areas managed by different servers, the proxy moves its connection for the client to the appropriate server. The proxy is aware of the regions of the world visible to the player, and thus connects to the servers holding those regions. It then filters out messages relating to areas not visible to the player, but passes along, in an integrated stream of updates, any messages from visible areas, regardless of the server from which they originated. In the implementation described here, the proxy is not fault-tolerant. If a given proxy fails in this implementation, those players connected to it are disconnected from the game, but the game continues to run without them. Implementation of a fault-tolerant proxy was considered beyond the scope of the project goals; however, either additional logic could be easily added to the game client to support automated failover, or some form of manipulation of network address or routing tables to the proxy pool could be used, as is done with Web servers, to provide failover.

The proxy can be thought of as a sort of multiplexer/demultiplexer. A single proxy allows a set of game servers and a set of game clients to communicate with one another while minimizing the necessary design and code modifications to either set. This means we can afford to have a simpler client, as the client does not have to be knowledgeable about the underlying network back end, nor does it need to know how to continually switch its connection across this back end. It also means that the back end can be totally inaccessible to the outside world, as long as the pool of proxies is accessible and these proxies can reach the back end. Finally, we can improve game-state updates. Proxies can handle the task of aggregating game-state information from all the various servers and ensuring that each client gets the exact information it needs, depending on its location and the visibility associated with that location.

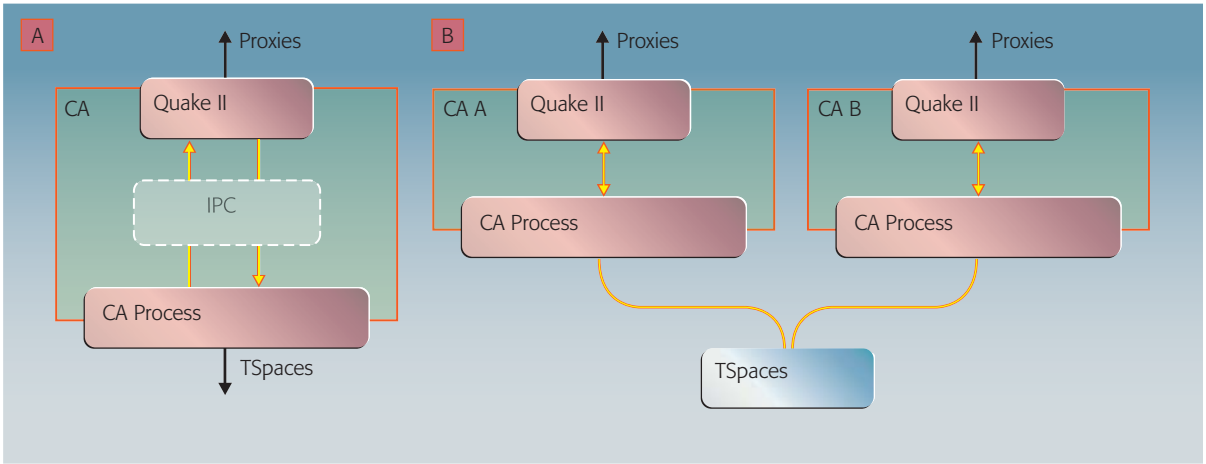

Figure 5
Communication: (A) Game server composed of two subcomponents, Quake II server and a CA connected by an interprocess communication (IPC) mechanism; (B) two game servers communicating by means of a TSpaces whiteboard.

Proxy alternatives

Alternative architectures that do not use a proxy are possible, though each has its own drawbacks. One alternative is to merge the proxy logic into the client itself. With the client responsible for managing its own connection switching, the need for a proxy pool would be eliminated. This would additionally eliminate the filtering of game updates sent to the client (passing only those from regions visible to the player), thus significantly increasing message traffic and bandwidth requirements between the client and the game servers. To alleviate this, servers could have additional logic added to perform some message filtering to clients, but this approach would increase the computational load on the game servers, reducing the number of players or the size of the world region each could host.

Another alternative is having a sticky server, that is, clients never switch connections; they connect to a server and stay connected. The game server itself performs the switching by allowing the client to tunnel through it as it moves through the game world. The drawback with this is that now a machine is acting as both a proxy and game server, when really they should be separate tasks. What happens when this game server goes down or the administrator wants to take it off the network? It is managing not only a piece of the game world, which can be moved to another grid node, but it is also acting as a proxy for some clients as well. This design is less robust, as clients may not have the logic necessary to switch proxies midstride, and if this is the case, shutting down a sticky game server will result in disconnections, even if the game state of that server is transferred.

Game server

When Quake II is deployed onto OptimalGrid, it runs as a child process of the OptimalGrid CA, which is a service that encapsulates and manages the responsibilities and resources of a grid node. In *Figure 5A* we illustrate how a game server is made up of two layers. The CA process—an application-specific Java process—is the lower layer and has a communication path to TSpaces, and with it, the rest of OptimalGrid. The upper layer is an instance of the Quake II server, which has a communication path to its clients via the proxy pool. The two layers of the game server communicate via an interprocess communications (IPC) mechanism that allows events and data to be sent and received. This mechanism couples the two processes written in different source languages and running in different runtime environments—the Java Virtual Machine (JVM**) and the native machine.

Interserver communication

Figure 5B illustrates the connection of two servers via TSpaces. When a game object crosses a server

boundary, that object state needs to be communicated from the original server S_1 to the new server S_2 very quickly and reliably. This transfer process happens in the following steps:

1. Game object P moves from game area A_1 through the game space on S_1, and its new game area A_2 is noted. Every time a game object moves into a new BSP-tree leaf cluster, the Quake II server notifies the CA of a possible need to transfer the game object.
2. Area A_2 is known to be on another server S_2. Therefore S_1 begins the dump sequence to remove P from its state and place it into TSpaces for S_2.
3. The Quake II process on S_1 removes P, serializes it, and sends it to its CA process on S_1 through an IPC mechanism.
4. The CA process places the serialized copy of P in a Java object, which is put into TSpaces.
5. The CA process of S_2 retrieves a tuple from TSpaces containing a Java object holding P.
6. From this Java object, the CA process extracts the serialized version of P and sends this version to its own Quake II process on S_2.
7. The Quake II process on S_2 deserializes P and places P back into the game world in its new position in A_2.

Quake II modifications

To give the Quake II server the mechanisms to manage only portions of the game world, when before it assumed control over it entirely, several extensions and modifications were necessary. Although we tried to avoid it, some minor changes to the Quake II client were also necessary, although these ended up being trivial and in the end did not alter much of the logic or behavior of the client:

- Entity serialization and deserialization midgame
- Map-file interpretation and partitioning
- Packet size maximum reached; extended the upper limit of the client packet size
- Game frame numbers translated and rewritten in proxy
- Packet sequence numbers translated and rewritten in proxy
- Game updates constructed in game server on a per-area basis as well as a per-client basis
- Game updates about areas from game servers routed by proxy to applicable clients

Proxy state

The proxy needs to maintain some routing information as its current state to properly route packets between the game servers and game clients. Routing a game packet is then a simple function of this state. After examining what was required to route the Quake II protocol, we found that several tables (lists of associations) needed to be kept current to route game traffic between clients and servers.

For the game updates that are created and delivered for each simulation frame of the game world, we need client-to-area mapping M_{CA} and area-to-area mapping, M_{AA}. For client messages that provide player input and commands to their corresponding game servers, we need client-to-server mapping, M_{CS}.

M_{AA} is static and an inherent property of the map and its current partitioning into swappable units. These units (which the proxy calls areas) are then distributed as responsibilities among the game servers and can later be traded between them in the process of load balancing. M_{AA} is unique because this part of the proxy state is static. The other mappings needed for routing are subject to change throughout the course of the game.

We take a common approach to keeping these mutable mappings up to date by putting their continually changing pairings in each update that is sent from the game servers to the proxies. For example, to keep M_{CA} up to date, we allow the servers to tag each client-specific message with the client's current area location. This avoids the need for explicit updates about clients' area changes, which tend to happen often as the clients transverse the virtual game world, but it also sidesteps another issue: all the traffic coming directly from the game servers to the proxy is, by default, unreliable. Because the datagrams that make up these messages can be lost by the underlying network stack, we would have to invent or adopt some kind of reliability layer to prevent our updates from getting lost. By tagging every client update with the client's current area, we avoid this problem.

The other dynamic mappings are maintained this way as well. As the flow of game traffic comes into the proxies from the game servers, the game servers prefix a header for the proxy with pieces of the current state of the proxy state mappings. The actual data that appears in the header corresponds to the

type of the game update packet being sent. For client-specific packets, client-routing data is prefixed. For area-specific updates, area-routing data is prefixed. The overhead of this data is minimal (eight bytes), but it allows the proxy state to remain current at all times and to cope with any disruptive properties of the underlying unreliable UDP, which does not include error detection or recovery/replay.

The state of each game entity is sent to every client that needs it in each frame of the game. This enables the proxies and modified game servers to accomplish the task of making a once-centralized world that is now split across many servers acting in concert.

Global views

Once the system is running, observing exactly what is going on from some global world view is a desirable administrative function. Without such a view, the only graphical view an administrator has of the game world is through a game client, and the only view of the game world's current occupation of the grid is by snooping about the communications between the grid components. What is needed is a snapshot of the state of the grid for a developer or administrator and a conceptual view of a working grid system for a newcomer.

We accomplish this by aggregating the problem state and responsibility of each grid node and presenting this aggregation graphically. This global viewing client, called the console or grid-eye view (and labeled *GEV* in Figure 2), provides this function. Using the console, we can see what, if any, responsibility has been given to each node in the running system, and we can see the human players or game bots active in the system and their location in the game world. From this view, we collapse many layers of abstraction: from physical machines, to grid services, to problem pieces, to game notions like map sections and players. Furthermore, our console gives us the ability to issue commands and query statuses for each node in the system. This includes the ability to dynamically move the game-world pieces between the participating grid nodes. An explanation of the supported operations is given in the next section, "Load balancing."

LOAD BALANCING

The architecture for Quake II on a grid thus far has not dealt with dynamically reconfiguring the use and number of servers in response to system load. In our architecture, OptimalGrid CAs are responsible for starting and stopping Quake II servers and for moving player and game objects between servers. There remains one more important feature provided by CAs: load balancing.

In the OptimalGrid core design, each CA is assigned a task set or VPP holding OPC collections. In Quake II terminology, an OPC collection represents an area in the Quake II world map, and a VPP represents a region composed of connected areas.

■ Load-balancing operations occur on demand as the APM identifies a suboptimal distribution of load across the grid. ■

In the OptimalGrid architecture, load balancing is achieved by migrating an OPC collection from the VPP on one CA to another. In the case of Quake II, this operation translates to adding or removing areas from the region controlled by a particular server. Entities in a migrated area are dropped by the server that formerly owned that area and are picked up by the server that now assumes responsibility for that area, similar to the operation that takes place when an entity crosses a region boundary.

Originally, OptimalGrid was designed to support scientific applications, which typically follow a synchronous model of communication. This synchronous mode of operation as supported by OptimalGrid is based on stages of computation; that is, no communication occurs between the CAs, followed by a stage of information exchange. This information exchange is required for the next stage to begin, providing a natural synchronization point for the grid application. OptimalGrid uses this feature to collect performance data and redistribute the workload accordingly. Quake II, on the other hand, is an asynchronous application. Interaction between different CAs is predicated on user interaction (entities moving from one region to another), which does not occur at regular intervals and does not involve all the CAs at the same time. We therefore must artificially synchronize the various CAs when reconfiguration occurs.

Synchronization is required to maintain consistency of the application state. To demonstrate this consideration, let us examine a simple scenario in which one player P_a is throwing a hand grenade at a second player P_b. The two players are located in two separate regions of the world map controlled by two different servers, S_a and S_b respectively. At this instant, the APM has decided to balance the system load and the problem is being repartitioned. It so happens that the OPC collection in which player P_b is located is now being reassigned to a third server. Let us assume that P_a's grenade in the original single-server setting would indeed hit P_b. In the case of a grid, without additional synchronization, the grenade may or may not hit P_b, depending on timing (i.e., a race condition exists). If, at the time when the grenade crosses the boundary of S_a's region, P_b's area is already assigned to the new server S_c but the player P_b himself has not yet been transferred from S_b to S_c, the grenade will appear to have missed player P_b. If, on the other hand, the grenade crosses S_a's boundary either before the transfer of P_b's area occurs or after it is completed, the grenade will indeed appear to have hit P_b, as in the single-server case. Such a condition is obviously unacceptable to Quake II users.

We have chosen to implement synchronization of the servers by freezing entities that take part in an inter-region interaction that involves a region which is being migrated for as long as the migration operation is in progress. This ensures that any additional inter-region activity is either wholly completed before the migration takes place or is delayed until after all the regions and their entities are picked up by their newly assigned CAs. Obviously, this is a trade-off solution that favors consistency over real-time performance. However, because the Quake II clients are designed to operate in an environment in which communication between the server and client may be intermittent and because the client maintains a map of the complete world, the client can interpolate between the world state updates to smooth out these freezes to the user. This greatly mitigates the effects of the freezes on the end user.

The load-balancing operations could be implemented in terms of contraction and expansion operations on the regions controlled by the servers. When a server's region is expanded, it is reconfigured to take on responsibility for the new areas added to it.

Entities in that area are then migrated to the expanded server, and finally the server that previously controlled this area has its region contracted. This solution, however, is Quake II-specific, in contrast to our solution that is built around OptimalGrid. We have chosen to implement the load-balancing operations in a more transparent way as far as the server code is concerned. Instead of modifying the region controlled by an already active server, our scheme creates a new server (under its own CA) that is started with a configuration appropriate for controlling the newly modified region. Then, entities in the region in the existing server are migrated to the new server. Finally, any superseded servers, which now control no entities, are shut down.

This solution has higher overhead because of the need to start new servers, and it also requires better management and knowledge of the grid configuration because new servers are started on a different machine. However, this solution is more generic and does not rely on the ability of individual servers to be reconfigured on the fly. It is also much better from a game-play point of view as the client game play is not interrupted by expansion, contraction, or server replacement.

Figure 6 demonstrates a repartition operation. On the upper half of the figure, we see S_a and S_b, where S_a controls an area R3 that is about to be reassigned to S_b. To perform the repartitioning operation, two new Quake servers are allocated, $S_{a'}$ and $S_{b'}$. The configuration of the two new servers is the configuration that S_a and S_b would have, respectively, after the repartition operation is completed. S_a and S_b drop all the entities in the areas they controlled, and the new servers $S_{a'}$ and $S_{b'}$ pick up the entities. After the operation is completed, the old S_a and S_b are shut down.

Load-balancing operations occur on demand as the APM identifies a suboptimal distribution of load across the grid. Identifying the ideal load distribution in the general case is a hard problem. The CAs report computation and communication loads at regular intervals or on demand by the APM. The APM uses pluggable algorithms to balance the load

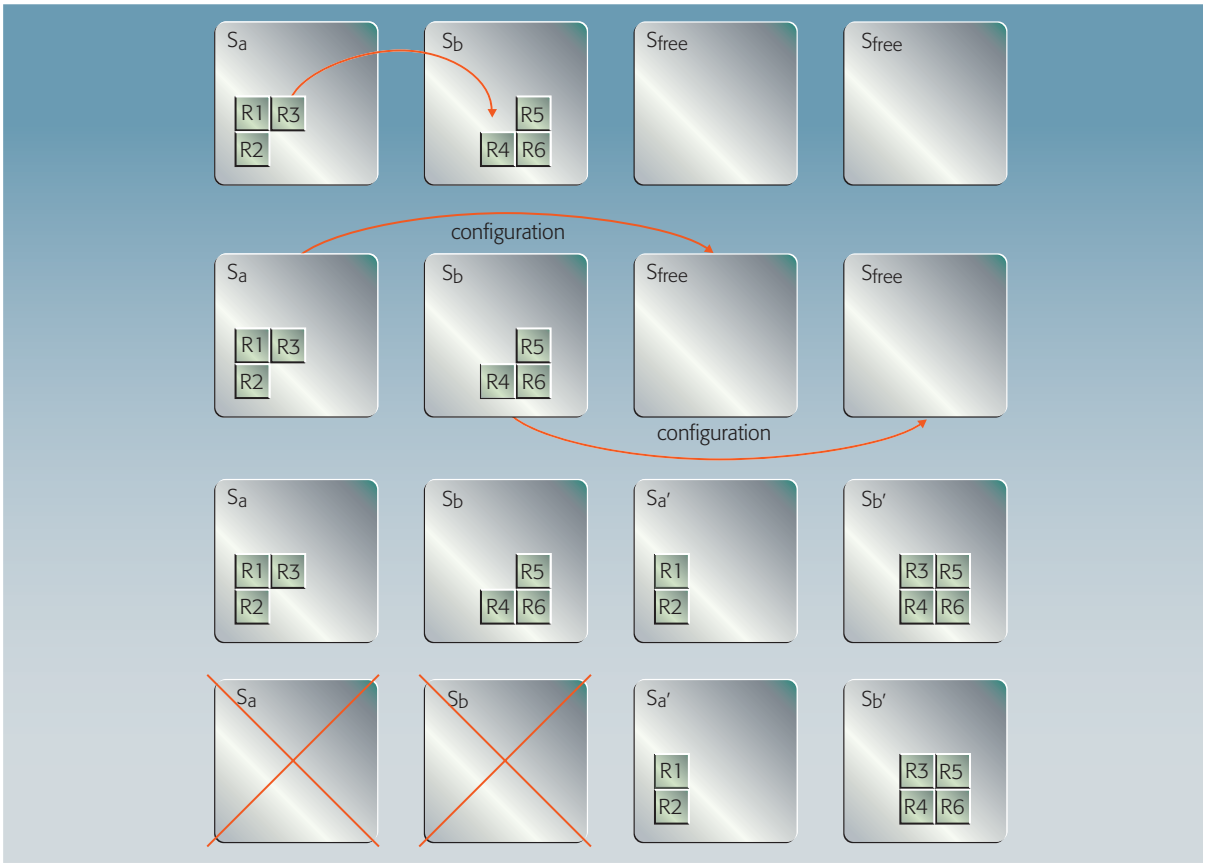

Figure 6
Repartition operation moving regions between servers

across available servers. It is then possible to implement policies based on any number of factors upon which optimization is desired: number of players on a server, the activity levels of players, the load on the CPU, and so on. In our example implementation, one policy we implemented was rebalancing based upon the number of players on a server: above a given threshold, the number of servers assigned regions expanded to two; below a given threshold, two servers were contracted into one.

The following explains the basic load-balancing operations supported by the OptimalGrid infrastructure:

- *Repartition*—Balance the load on a server by reassigning a region of the world map controlled by one server to another server. In practice, the two old servers are shut down, and two new servers are assigned the two new regions resulting from the repartitioning operation. This approach of using two new servers was chosen so that we can ensure full initialization and synchronization of the new servers before using them. Future versions will likely do away with this very conservative approach.
- *Expand*—Split the region set assigned to one server into two smaller region sets and reassign the two newly created regions to two new servers. As with the repartition operation, we followed a conservative strategy of not reusing the existing server as one of the servers after the split. It is unlikely that future versions will be this conservative.
- *Contract*—Merge two regions controlled by two servers into one. Reassign the newly created region to a new server. Once again the strategy of using a new server was chosen for this implementation.

- *Migrate*—Reassign the region controlled by one server to a new server. The old server is then shut down.

In the case of the Quake II application, the load on individual servers is directly related to the number of players within the control region of the server and is independent of the size of the control region itself. Therefore, the load-balancing mechanism used by the APM can be greatly simplified and based on thresholds set for the difference between the numbers of players in each region.

RUNTIME ENVIRONMENT

Instantiating Quake II on OptimalGrid is a carefully orchestrated workflow across the many computers on which the services that make up the system will be run. Unlike a simple server running on only one computer, Quake II on OptimalGrid has one or more TSpaces servers, one or more CAs, one or more proxies, one APM, and one problem builder. Additionally, one or more computers may be used to run bots. Each of these services has build and configuration issues that need to be handled.

We initially created a tool to assist in setting up and starting Quake II on OptimalGrid. This was intended to make it easy for users to start and stop the system. The first version of the tool was written in Perl and was directly tied to the Quake II on OptimalGrid application. The original version was written in a few days and met our basic and initial needs as developers. As Quake II on OptimalGrid began to be used by more and more people outside our immediate project team, the need to simplify the configuration process for both nonexpert users and the research team motivated us to revisit the tool. The second version was more general. It was written in Object-Oriented Perl and supported an external XML (Extensible Markup Language) configuration file.

The Quake II on OptimalGrid runtime system was designed to work on Linux**. OptimalGrid can run on any Java-enabled platform, but the Quake II modifications made by the research team were done only to the Linux version of the Quake II server. This reduced the complexity of controlling the runtime environment because only Linux needed to be supported. The source for Quake II on OptimalGrid consisted of both the Java code of OptimalGrid, the C code of the Quake II server, and the proxy we developed.

First runtime system

The first runtime system, written in Perl, followed a hub-and-spoke model, where all the compilation, configuration, and packaging into tar files was done on the hub, then pushed (using the Linux `scp` tool) to the appropriate node on the network, and then unpacked. The services on each node were started and stopped by means of a remote invocation of a copy of the script on the hub by using the Linux `ssh` tool. The spoke version would then directly invoke the executable to start the service or issue the Linux `kill` command to stop it. In this original version, the specific machines that ran CAs, and thus servers, were hard-coded into the master script. This script was then used during the build and bundling process to select the map partitioning that would be used. Thus, in order to change either the specific server assignment or the number of servers, the source file of the script had to be edited and completely rebuilt, followed by a complete redistribution and unpacking of the system executables. Even simple changes in the configuration were slow and painful to make.

A consequence of doing all the builds on the one master node is that the nodes that run the compiled executables must have compatible runtime libraries. Linux systems are patched regularly to incorporate fixes and changes, particularly the GNU C compiler libraries. For this reason, the hub node used for the builds needed to be identical to the spoke nodes, which would run the binaries. Although this solution met the immediate needs of the research team, its ongoing use revealed areas where usability could be improved.

Second-generation runtime system

As use of the first runtime system grew beyond the immediate research team, there were a number of areas to improve:

- *Configuration editing*—Changing configuration should only require the editing of one file. The configuration file should be easy to edit, and it should be programmatically verifiable for correctness. This was done by isolating all the configuration parameters in one XML file: nodes, node assignments, service options, and so forth.

XML provided the ability to programmatically verify the configuration file. This allowed us to implement a syntax checker for the configuration file. A detailed semantic checker could have been created as well by using a document type definition and a validating parser, but this was never implemented.

- *Configuration changes*—The runtime system should only need to move the updated configuration file, and possibly any new code required, to each node, instead of doing a complete rebuild on the hub node, followed by a complete replacement of any existing code on the spoke nodes. This requirement was met by changing when and where the OptimalGrid problem configuration was performed. In the first version, this was done when distribution tar files for each service were compiled and built. The tar files for the APM and the CAs included the explicit Quake II map partitioning for the specific number of servers for which the system was being built. Thus, every change to the number of servers required a rebuild and redistribution of tar files. This was avoided by making several changes. First, the build process was changed so that it did not require any specific knowledge of which servers were being used or their number or roles. The configuration for a specific setup was moved into the XML configuration file. Second, the problem builder process was modified to be run as part of the launch sequence of the system. This allowed the details of the system to be specified at launch time instead of build time. Another benefit of this was that once the executable for a service had been installed on a node, only the configuration file needed to be updated when different system setups were made.
- *Build*—The runtime system should be able to run on different Linux systems. The requirement that all the nodes be clones of the hub, or at least have compatible runtime libraries, was very limiting. The restriction to Linux only was not a problem, but the restriction of all nodes being the same distribution and version of Linux was. For example many users wanted to run our Quake II on OptimalGrid system, but the collection of systems available to them on which to run would include a heterogeneous mixture of Linux versions and distributions, such as Red Hat** 7, Red Hat 8, Red Hat 9, and SUSE** Linux to name a few. We solved this by moving the compilation process of C code from the hub node onto each of the spokes. This did mean that when a service was set up on a node for the first time, it was necessary to wait while the C portions of it compiled, but this was a one-time penalty. Subsequent starts would reuse the previously built binaries.
- *Service assignment and status*—One of the limitations of the original system was determining the service assignments to nodes as coded in the script file. To discover them, users would have to look at the source of the Perl script. For non-Perl users, this was not very friendly. Another limitation was that once the system had been launched, there was no easy way to see the status and assignment of all the services. We solved this by introducing two features into the system. The first feature can parse the XML configuration file and produce a user-friendly summary of the system configuration. The second feature dynamically queries the status of each service on each node as laid out in the XML configuration file and returns its current status. These two features provide easy understanding of the system's configuration and real-time status.

The new runtime system was written in Object-Oriented Perl. A set of utility classes were created for performing interaction with the underlying operating system and for data management. A base class representing a generic service was created, with subclasses created for each of the services: APM, problem builder, CA, proxy, and bots. The use of object-oriented design allowed for easy extension of the system for use in other OptimalGrid applications and beyond. New services could be added quickly by extending the base class and implementing tasks specific to the service, such as building, configuring, starting, stopping, and asking for status.

Another change in the design affected how the hub issued commands on the spoke nodes. In the original version, the script on the hub was duplicated on each spoke. It would then call its copy of itself, using a different subroutine as its entry point. This is how the script would determine whether it was running on the hub or a spoke. The change we made was to create specific scripts for the hub and for the spokes. Following object-oriented design principles, we then created classes and subclasses as needed for the two sides of the conversation. The spoke script would instantiate the appropriate Perl object class for the service on the node, which

would then follow the configuration in the XML file and perform the requested action on the service: start, stop, status, and so forth.

Hub and spoke conversations

One of the limitations we encountered was in detecting the results of actions the hub node asked the spokes to perform. The `ssh` session used to connect with the spoke would only give the return code of the last command to execute as an integer when the `ssh` session was completed. Additionally the messages to the `stdout` stream could be captured, but they were only a stream of characters without context. The `stderr` messages for remotely executed tasks were difficult to capture in Perl—not impossible, just difficult—but they too lacked context. This made it difficult for the script on the hub to do anything more meaningful than just dump the messages to the console. A considerable amount of mostly meaningless text was displayed when launching the system on 20 nodes. The solution was to create an XML messaging object that was used by the hub to send commands to the spokes, and for the spokes to send all output. This allowed the code on the spokes to produce output as XML messages and hold the resulting output from the scripts and commands on the spoke, while retaining the context of the text along with the return code associated with them. With this additional knowledge, the hub script could then be extended to perform more intelligent and useful actions than was possible in Version 1.

DEBUGGING

The task of parallelizing Quake II with OptimalGrid presented the significant challenge of debugging the large distributed application. In a simple client/server application, there are only two parts to consider when tracking down problems: the client and the server. Behavior of both can be directly observed through application graphical user interfaces, consoles, and the log files of each. Quake II on OptimalGrid, by comparison, is much more complex to debug.

The OptimalGrid services—CAs, APM, TSpaces, and problem builder—are distributed across several systems connected together to form the OptimalGrid runtime. The pieces produce large amounts of debugging information across different nodes and servers. This information must be integrated to present a useful overall view of the application state. The addition of the Quake II servers, Quake II clients, proxies, and proxy filters increases the number of sources and also introduces new interconnected relationships (Quake II servers to CAs, Quake II clients to Quake II servers, etc.) that must be integrated for effective debugging of problems. This resulted in our creation of several ways to improve access to debugging information and to make viewing it easier.

Logs

Each service and Quake II component produces some form of log output to help in debugging. In a typical test done during development, nine grid nodes would be allocated. One node would run the APM, TSpaces, problem builder, proxy filter, and proxy, resulting in five logs. Six nodes would run CAs and Quake servers, adding two logs per node (two would be running CAs waiting for use as needed and adding one log for the CA, and one log each if a server was started). This adds up to 17 log files across the nine nodes if load balancing is not active, and up to 19 if it is. The problem grows as the scale of the test grows.

Accessing these numerous independent logs was tedious, requiring numerous logins. Our solution to this was to add log access to our launch and distribution system in the operating environment. The launch systems knew which grid nodes were used, what roles they were assigned, and where each service log file was on local disk. The ability to log in and access the files was relatively easy to add. The problem remaining was how to display the logs themselves. Two options were provided to developers. The first was a dedicated window per log file, created on the developer's workstation. This was updated in real time as the log file was changed on the remote grid node. The second was a merged output of all the log files being monitored. This output would merge the stream of messages from each monitored log file and show them together in a single window. It was also possible for specific services to be monitored, for example, only CAs if that was all the developers needed. This was made available to the developer as a simple one-line command.

Log file content detail was also selectable, allowing developers to increase the level of detail for

particular services, either for all instances of the service or instances on specific nodes. The combination of log level control and easy one-command access greatly improved the ability for developers to quickly monitor and locate problems.

Messages

A second form of output by the OptimalGrid system is messaging through the TSpaces communication system. Embedded in each OptimalGrid service is a debug class that can generate and send message objects. Objects can include simple events generated by the services up to the Java event objects generated by fatal exceptions. In addition to the more traditional logs in use, these objects are useful because clients can be easily written, making it possible to monitor the message space and actively filter which messages are displayed to the user. Filtering based on any attribute of the message object is possible: source, service, message content (error only, server errors, memory exceptions, etc.). Messages are accessible by a small footprint client we wrote for the task or from any Web browser.

PERFORMANCE OF QUAKE II AS AN MMOG

Several different experiments were done to validate the performance of the MMOG version of Quake II. The primary goal was to determine if our implementation satisfied the single most basic criterion: Was game play from the player's perspective any different when on a grid? Two tests are presented here that demonstrate we did indeed achieve this goal. Deeper, more robust performance testing and analysis is beyond the scope of this paper and belongs in separate work.

Player transfer latency

The first test was done precisely six weeks after coding began on the project. In this test, the modified Quake II client was made available to people inside the IBM Research Division who were invited to play the game. The invitation eventually was sent to other IBM employees around the world. This early test was designed to validate the code itself (determine that the servers would not crash due to coding errors) and to demonstrate that the Quake II on OptimalGrid distributed system could deliver adequate performance for an FPS game. The critical factor in this performance evaluation was the player transfer latency or the time required to pull a moving object or player out of one server and reinsert the object or player into a destination server. The player transfer latency could not exceed the Quake II world update time, which is fixed at 100 ms. If the player transfer latency exceeded 100 ms, then the player would notice a distinct jitter or interruption when moving around the world among servers. Note that the player transfer latency is not correlated with the overall game play latency, which depends on many factors, including network infrastructure, network traffic, and raw server speed. In our tests, the normal game play latency was quite good, and game play within a server was indistinguishable from game play on a single (nondistributed) Quake II game.

Figure 7A shows the player transfer latency measured in the second of two separate tests of the Quake II system. Note that in reporting player transfer latency it is not sufficient to report the average latency achieved. A histogram of the data more accurately represents the game play as the tail of the distribution reveals how often (or if) a critical threshold time or game-world update time might be exceeded.

The first performance test used 30 servers and a 100-Mb Ethernet infrastructure for the servers. In this first test, the average player transfer latency was approximately 150 ms, and game play across servers had a noticeable jitter or delay. Subsequently, we reexamined some implementation choices, making several changes.

We conducted a second test using the updated code and improved server hardware. The second test results, shown in Figure 7A, reflect the performance when the system was moved to a server cluster of dual Intel Xeon** processor nodes with a Gigabit Ethernet infrastructure. The game was distributed over 12 servers. In this test, the average player transfer latency was less than 70 ms, which is less than the Quake II server 100-ms update frequency. In this test there was no obvious delay when players moved between servers, meaning we had achieved the core goal that players should get as good a playing experience in regard to performance on the grid as they did on a single server.

Between these two tests we made a change to the algorithm used in the problem builder to perform the initial partitioning of the game map. This was one of the most significant factors in the perfor-

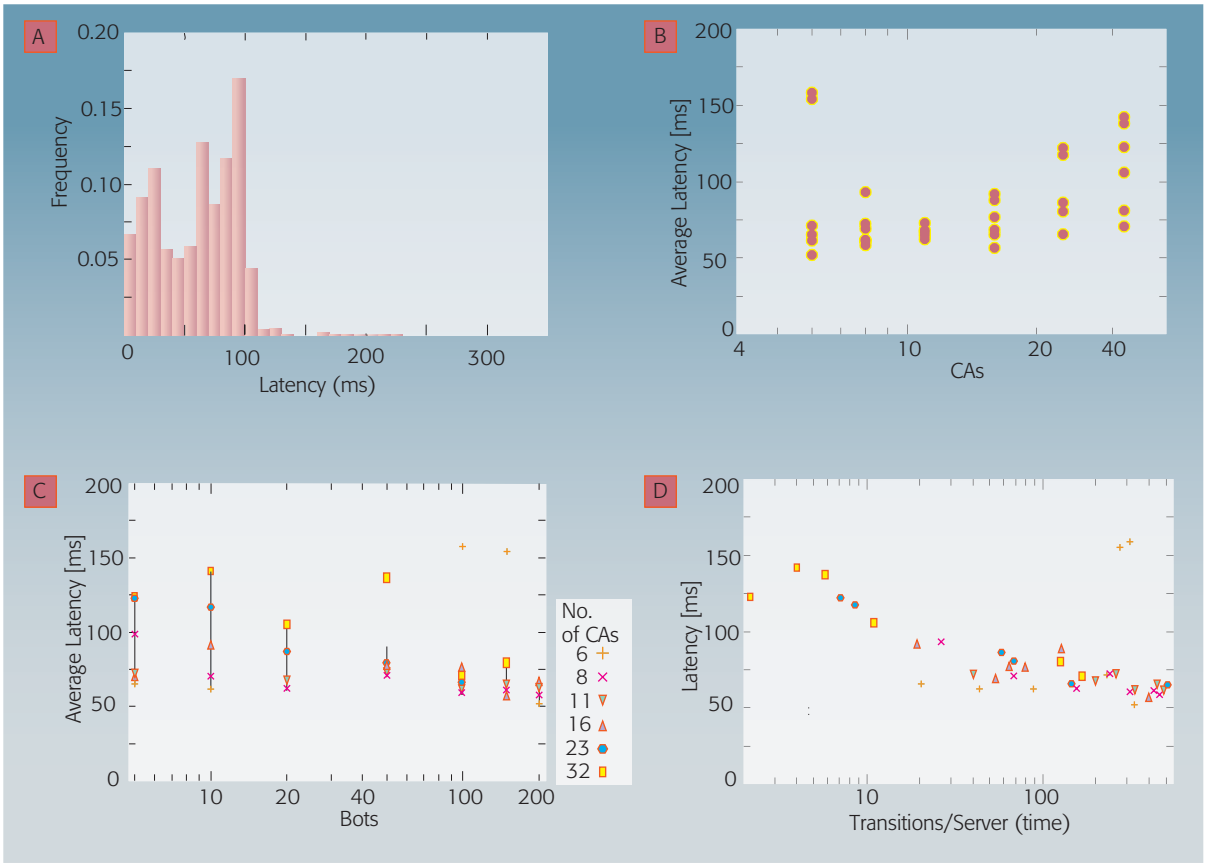

Figure 7
Player transfer latency: (A) Frequency of player transfer latency occurrence; (B) player transfer latency as a function of the number of game servers (CAs); (C) player transfer latency compared with the number of bots; (D) player transfer latency as a function of per-server transfer frequency. (The unit time interval was four minutes.)

mance improvement seen in the second test. In particular one aspect we did not fully consider in the original partitioning scheme was the 3D nature of the world. The one poor choice made in the original algorithm was to occasionally partition stairways too finely. This resulted in players moving between servers with each step on the stairway.

As we have experimented with this MMOG version of Quake II, we have subsequently made other improvements to the system. These were individually minor, making it difficult in a paper of finite size to discuss, but the sum of their contributions was significant. It is worth noting however, that even in the early implementation profiled here, we were able to demonstrate adequate performance for an FPS game.

Optimal map partitioning

For any infrastructure and system design, the efficiency of the map-partitioning algorithm has a significant effect on overall system performance. In simple terms, a partitioning algorithm that lowers surface-to-volume ratio lowers the rate at which players cross server boundaries for a given map design. This fact is evident in *Figure 7B*, which shows latency compared with the number of game servers (CAs). For control purposes, this data was acquired with a constant number of 200 client-side Quake II bots running on the 100-Mb Ethernet cluster. The data demonstrates that for the map and partitioning algorithm in use, the system is most efficient (and performance most consistent) when the game world is divided across 10–12 servers.

Scalability

Ultimately, it is important to consider how total game load limits overall system performance. In our live-play tests, we had difficulty finding sufficient numbers of people who could play simultaneously to stress our test grid. To study the system at even

higher loads, we measured the latency as a function of the number of Quake II client-side bots, where the number of bots was varied between one and 200. We could run up to five bots per server, which limited the range of this test. The latency of one to 200 bots running on six to 30 servers is shown in *Figure 7C*. There is a large scatter in the data, but analysis reveals an important scaling function. If the same data is plotted as player transfer latency against the number of server crossings per server per second (which increased with the number of servers and the number of players or bots), then all of the data collapses onto the single curve shown in *Figure 7D*.

Analysis

Somewhat surprisingly, the system performance initially improves as the load on the communication system is increased. This improvement is likely due to two factors. At very low load, our original Quake II code ran open loop with all CAs continually polling the whiteboard servers to detect new players arriving. When game traffic was low this was highly inefficient, as requests for data were frequent but returned nothing (there was no data). This situation was later improved by modifying the system so that if a query for arriving players returned null, the CA would enter a *wait to take* state (registering for a callback on the TSpaces server). In this state, if a player arrives, in response to the callback the server immediately enters a multitake loop and processes arriving players continually until a query returns null, in which case it returns to the *wait to take* state. This design is much better given the dynamic nature of game play (with varying levels of activity of different servers).

The second reason for the improvement in performance at higher loads may be understood in terms of the multitake operation itself. In periods of intense traffic when many players are crossing a common server boundary in the same 100-ms interval, these player objects are written to and read from a single whiteboard in a single multitake or multiwrite operation. At high loads, the number of player server crossings can increase with no increase in the number of messages or transactions and only a slight increase in message size. This predicts a nearly linear improvement in performance as load increases. Of course at very high load, other factors should cause the performance to decrease, but we were unable to observe this, given our limit of 200 bots in the experiment. This regime may be explored in future studies.

CONCLUSION

MMOGs are an emerging form of online entertainment. While the common approach of addressing the demands of MMOGs can be done through the design and implementation of specialized MMOG-specific game server engines, we have demonstrated that existing engines, such as Quake II, can be extended to become dynamically scalable engines capable of meeting these same demands without compromising game play or game design. This was possible through the use of intelligent autonomic middleware, such as IBM OptimalGrid, which handled the decomposition of the game world into connected regions capable of being hosted on different servers, and subsequently the reintegration of the game world messages, enabling clients to see the distributed virtual world as one that was single and unified.

■ The results of our performance tests showed that the resulting MMOG Quake II engine was capable of meeting game-play needs. ■

Autonomic load balancing, server expansion, server consolidation, and server start/stop control were added, along with a simple-to-use provisioning system, making the resulting system more than a trivial example. Instead, it is an advanced implementation that showcases how OptimalGrid can be used in a practicable way to add features that are vital in production-level commercial deployments of MMOG engines servicing paying customers. The results of our performance tests showed that the resulting MMOG Quake II engine was capable of meeting game-play needs, providing a seamless world, and hiding from the players the reality of running the world across more than one server. Finally, this extension of the Quake II engine to MMOG status was done while strictly following the constraint of making only minor changes to the original Quake II server. The solutions to the challenges faced in this work are reusable features that can be applied, without redesign of the game engine core, to add dynamic load balancing, dynamic scaling, support for distributed servers, and

advanced execution management to other existing game engines, making it possible to give them new life as MMOGs.

The changes we made to the Quake II open-source game engine and client, and the OptimalGrid middleware to run them, is available on the Web from the IBM alphaWorks*[21] site.

FUTURE WORK

This paper focused on the creation of an MMOG game engine beginning with an existing single-server online engine. In particular, the engine chosen was an FPS game. Another popular type of game for MMOGs is the role-playing game, often called MMORPG, which is typically much longer running than FPS games, with both characters and the game world having significant and evolving amounts of state information. Such state information is maintained in the game server memory, with copies of it backed by network-connected databases. This introduces a new set of issues, such as how to use a grid to enhance the performance and scalability of the distributed database servers used by the game engine.

The use of BSP trees in Quake II to represent the game world, wherein the world is represented as a graph with leaf nodes being visible vertices, made the choice of mapping these to OptimalGrid OPCs the obvious choice. This—together with the fact that the primary limit faced by the Quake II engine when hosting a large number of players is the ability of a server to process player-client messages and updates for all the regions of the map—made the partitioning of the Quake II engine along map regions the most appropriate one. This choice however may not be the best to apply to different game engines that face different resource limitations when required to host large numbers of players. The ability to partition and distribute the computational needs of these alternate engines would require analysis of the particular design and behavior of the engine to make the appropriate partitioning decision. One can imagine a game in which the game world would be hosted in full on each server, and instead, its computationally expensive task is game physics. In this case, it would be best to partition and distribute physics calculations for the game regions on the grid, with the physics calculations and game world areas forming the interconnected graph, instead of the graph of connected visible areas used for Quake II. This different partitioning and management is still possible with OptimalGrid, but would require that a different OPC and problem builder component be developed.

CITED REFERENCES

1. Emergent Game Technologies, http://www.emergentgametech.com/press_1.html.
2. M. Oliveira, J. Crowcroft, and M. Slater, "An Innovative Design Approach to Build Virtual Environment Systems," *Proceedings of the ACM International Conference Proceeding Series, 9th Eurographics Workshop on Virtual Environments*, Zurich, Switzerland (2003), pp. 143–151.
3. A. Shaikh, S. Sahu, M. Rosu, M. Shea, and D. Saha, "An On Demand Platform for Online Games," *IBM Systems Journal* **45**, No. 1, 7–20 (2006, this issue).
4. K.-W. Lee, B.-J. Ko, and S. Calo, "Adaptive Server Selection for Large Scale Interactive Online Games," *Proceedings of the 14th International Workshop on Network and Operating Systems Support for Digital Audio and Video, International Workshop on Network and Operating System Support for Digital Audio and Video*, Cork, Ireland (2004), pp. 152–157.
5. E. Frécon and M. Stenius, "DIVE: A Scalable Network Architecture for Distributed Virtual Environments," *Distributed Systems Engineering Journal* **5**, No. 3, special issue on Distributed Virtual Environments, pp. 91–100 (September 1998), http://www.iop.org/EJ/abstract/0967-1846/5/3/002.
6. Wu-chang Feng and Wu-chi Feng, "On the Geographic Distribution of On-line Game Servers and Players," *Proceedings of the Second Workshop on Network and System Support for Games*, Redwood City, CA (2003), pp. 173–179.
7. A. Tveit, O. Rein, J. V. Iversen, and M. Matskin, "Scalable Agent-Based Simulation of Players in Massively Multiplayer Online Games," *Proceedings of the Eighth Scandinavian Conference on Artificial Intelligence (SCAI2003)*, Bergen, Norway (November 2003), http://www.idi.ntnu.no/~amundt/publications/2003/zereal.pdf.
8. A. F. Seay, W. J. Jerome, K. S. Lee, and R. E. Kraut, "Project Massive: A Study of Online Gaming Communities," *Conference on Human Factors in Computing Systems (CHI'04)*, Vienna, Austria (2004), extended abstracts on human factors in computing systems, pp. 1421–1424.
9. T. J. Lehman and J. H. Kaufman, "OptimalGrid: Middleware for Automatic Deployment of Distributed FEM Problems on an Internet-Based Computing Grid," *Proceedings of the IEEE International Conference on Cluster Computing*, Hong Kong, China (2003), pp. 164–171.
10. Half-Life, Valve Corporation, www.valvesoftware.com.
11. Half-Life, Sierra Entertainment Inc., http://half-life.sierra.com/.
12. Quake, id Software, http://www.idsoftware.com/games/quake/quake3-gold/.

13. OptimalGrid: Autonomic Grid Systems, IBM Research, http://www.almaden.ibm.com/software/ds/OptimalGrid.
14. S. W. Golomb, "How to Number a Graph," *Graph Theory and Computing*, R. C. Read, Editor, Academic Press, New York (1972), pp. 23–37.
15. Project OGR, www.distributed.net/ogr.
16. Kingpin, developed by Xatrix Entertainment Inc., published by Interplay, Inc., 1999.
17. Soldier of Fortune, Studio—Raven Software Corp., published by Activision, Inc., www.activision.com.
18. D. Gelernter and A. J. Bernstein, "Distributed Communication via Global Buffer," *Proceedings of the ACM Principles of Distributed Computing Conference,* Ottawa, Ontario, Canada (1982), pp. 10–18.
19. D. Gelernter, "Generative Communication in Linda," *TOPLAS* **7**, No. 1, pp. 80–112 (1985).
20. TSpaces, IBM Research, http://www.almaden.ibm.com/cs/tspaces.
21. alphaWorks, IBM Corporation, http://www.alphaworks.ibm.com.

Accepted for publication June 14, 2005.
Published online January 19, 2006.

Glenn Deen
IBM Research Division, Almaden Research Center, 6560 Harry Road, San Jose, California 95120 (glenn@almaden.ibm.com).
Mr. Deen joined IBM in 1989. He is a researcher and the OptimalGrid project leader at the IBM Almaden Research Center. His current work is in the areas of distributed computing and healthcare informatics. In the past he has been involved in such areas as data visualization, distributed storage, and distributed security policy and architecture.

Matthew Hammer
IBM Research Division, Almaden Research Center, 6560 Harry Road, San Jose, California 95120 (hammer@upl.cs.wisc.edu).
Mr. Hammer is a computer science Ph.D. candidate at the Toyota Technological Institute in Chicago specializing in programming language theory and implementation. While an undergraduate at the University of Wisconsin, he worked at the IBM Almaden Research Center on various grid computing projects including OptimalGrid and the Extreme Blue project GameGrid.

John Bethencourt
IBM Research Division, Almaden Research Center, 6560 Harry Road, San Jose, California 95120 (jbethenc@andrew.cmu.edu).
Mr. Bethencourt is a computer science Ph.D. candidate at Carnegie Mellon University with a research focus in applied cryptography and systems security. He has been involved in several grid computing projects, including the Condor project at the University of Wisconsin (where he was an undergraduate) and the OptimalGrid project at the IBM Almaden Research Center.

Iris Eiron
IBM Research Division, Almaden Research Center, 6560 Harry Road, San Jose, California 95120 (iriss@us.ibm.com).
Ms. Eiron is a researcher at the IBM Almaden Research Center. She joined IBM in January 1998 after receiving an M.S. degree in computer science from the Technion, the Israeli Institute of Technology. She worked for the IBM Israeli Research Lab for three years. In December 2000 she joined the Almaden Research Center. Ms. Eiron's current interests include development and implementation of a national healthcare infrastructure.

John Thomas
IBM Research Division, Almaden Research Center, 6560 Harry Road, San Jose, California 95120 (jgthomas@us.ibm.com).
Mr. Thomas is a researcher in the department of Computer Science at the IBM Almaden Research Center. He is a Java developer for IBM. Mr. Thomas was previously one of the lead programmers for the IBM Almaden TSpaces project and a member of the OptimalGrid Project at the Almaden Research Center.

James H. Kaufman
IBM Research Division, Almaden Research Center, 6560 Harry Road, San Jose, California 95120 (kaufman@almaden.ibm.com).
Dr. Kaufman is the manager of the Healthcare Informatics project in the department of Computer Science at the IBM Almaden Research Center. He received a B.A. degree in physics from Cornell University and a Ph.D. degree in physics from the University of California at Santa Barbara. During his career at IBM Research, he has made contributions to several fields ranging from simulation science to magnetic device technology. His scientific contributions include work on pattern formation, conducting polymers, superconductivity, experimental studies of the moon illusion, and contributions to distributed computing and grid middleware. Dr. Kaufman is a Fellow of the American Physical Society. ■