Jim Waldo, Sun Microsystems Laboratories

SCALING
in games & virtual worlds

ONLINE GAMES AND VIRTUAL WORLDS HAVE FAMILIAR SCALING REQUIREMENTS, BUT DON'T BE FOOLED: EVERYTHING YOU KNOW IS WRONG.

I USED TO BE LIKE YOU.

I used to be a systems programmer, working on infrastructure used by banks, telecom companies, and other engineers. I worked on operating systems. I worked on distributed middleware. I worked on programming languages. I wrote tools. I did all of the things that hard-core systems programmers do.

And I knew the rules. I knew that throughput was the real test of scaling. I knew that data had to be kept consistent and durable, and that relational databases are the way to ensure atomicity, and that loss of information is never an option. I knew that clients were getting thinner as the layers of servers increased, and that the best client would be one that contained the least amount of state and allowed the important computations to go on inside the computing cloud. I knew that support for legacy code is vital to the adoption of any new technology, and that most legacy code has yet to be written.

But two years ago my world changed. I was asked to take on the technical architect position on Project Darkstar, a distributed infrastructure targeted to the massive-multiplayer online-game and virtual-world market. At first, it seemed like a familiar system. The goal was to scale flexibly by enabling the dynamic addition (or subtraction) of machines to match load. There was a persistence layer and a communication layer. We also wanted to make the programming model as simple as possible, while enabling the system to use all the power of the new generations of multicore chips that Sun (and others) were producing. These were all problems that I had encountered before, so how hard could these particular versions of the problems for this particular market be? I agreed to spend a couple of months on the project, cleaning up the architecture and making sure it was on the right track while I thought about new research topics that I might want to tackle.

SCALING
in games & virtual worlds

The three months have turned into two years (and counting). I've found lots of new research challenges, but they all have to do with finding ways to make the environment for online games and virtual worlds scale. In the process, I have been introduced to a different world of computing, with different problems, different assumptions, and a different environment. At times I feel like an anthropologist who has discovered a new civilization. I'm still learning about the culture and practice of games, and it is a different world.

EVERYTHING YOU KNOW IS WRONG

The first thing to realize in understanding this new world is that it is part of the entertainment industry. Because of this, the most important goal for a game or virtual world is that it be fun. Everything else is secondary to this prime directive. Being fun is not an objective measure, but the goal is to provide an immersive, all-consuming experience that rewards the player for playing well, is easy to learn but hard to master, and will keep the player coming back again and again.

Most online games center around a story and a world, and the richness of that story and world has much to do with the success of the game. Design of the game centers on the story and the gameplay. Design of the code that is used to implement the game comes quite a bit later (and is often considered much less interesting). A producer heads the team that builds the game or the virtual world. Members of the team include writers, artists, and musicians, as well as coders. The group with the least influence on the game consists of the coders; their job is to bring the vision of others to reality.

The computational environment for online games or virtual worlds is close to the exact inverse of that found in most markets serviced by the high-tech industry. The clients are anything but thin; game players will be using the highest-end computing platforms they can get, or game consoles that have been specially designed for the computational rigors of these games. These client machines will have as much memory as can be jammed into the box, the latest and fastest CPUs, and graphics subsystems that have supercomputing abilities on their own. These clients will also have considerable capacity for persistent storage, since one of the basic approaches to these games is to put as much information as possible on the client.

The need for a heavyweight client is, in part, an outcome of the evolution of these games. Online games have developed from stand-alone products, in which everything was done on the local machines. This is more than entropy in the industry, however; keeping as much as possible on the client allows the communication with the server to be minimized, both in the number of calls made to the server and in the amount of information conveyed in those calls. This communication minimization is required to meet the prime directive of fun, since it is part of the way in which latency is minimized in these games.

Latency is the enemy of fun—and therefore the enemy of online games and virtual worlds. This is especially interesting in the case of online games, where the latency of the connection between the client and the servers cannot be controlled. Therefore, the communication protocol needs to be as simple as possible, and the information transmitted from the client to the server must fit into a single packet whenever possible. Further, the server needs to be designed so that it is doing very little, ensuring that whatever it is doing can be done very quickly so a response can be sent back to the player. Some interesting tricks have been developed to mask unavoidable latency from the player. These include techniques such as showing prerecorded clips during the loading of a mission or showing a "best guess" immediately at the result of an action and then repairing any differences between that guess and the actual result when the server responds.

The role of the server is twofold. The most obvious is to allow players to interact with each other in the context of the game. This role is becoming more important and more complex as these games and worlds become increasingly elaborate. The original role of the server was to allow players to compete with each other in the game. Now games and virtual worlds are developing their own societies, where players may compete but may also cooperate or simply interact in various ways. Virtual worlds allow users to try out new personalities; games let players cooperate to do tasks that they would be unable to complete individually. In both, players are finding that a major draw of the technology is using it to connect to other people.

The second role of the server is to be the arbiter of truth between the clients. Whether the client is running on a console or on a personal computer, control rests in the hands of the player. This means that the player has access to the client program, and the competitive nature of the games gives the player motivation to alter the

client in the player's favor. Even in virtual worlds, where there is only social competition, the desire to "enhance the opportunity" of the individual player (also known as "cheating") is common. This requires that the server, which is the one component that is not under the control of the players, be the arbiter of the true state of the game. The game server is used both to discourage cheating (by making it much more difficult) and to detect cheating (by seeing patterns of divergence between the game state reported by the client and the game state held by the server). Peer-to-peer technologies might seem a natural fit for the first role of the game server, but this second role means that few if any games or worlds trust their peers enough to avoid the server component.

CURRENT SCALING STRATEGIES

The use of the singular term *server* in the previous section represents a conceptual illusion of the system structure that can be maintained only by the clients of the game or world. In fact, any online game or virtual world will involve a large number of servers (or will have failed so miserably that no one either can or wants to remember the game or world). Using multiple servers is a basic mechanism for scaling the server component of a game to the levels that are being seen in the online world today. World of Warcraft has reported more than 5 million subscribers with hundreds of thousands active at any one time. Second Life reports usage within an order of magnitude of World of Warcraft, and there is some evidence that sites such as Webkinz or Club Penguin are even more popular. A single server is not able to handle such load, no matter how efficient the representation. Even if a single server could deal with this load, such a server would be far too expensive for the smaller loads that are encountered (sometimes by the same games or worlds) at times of low demand (or in parts of the product's life cycle when demand has decreased).

Having multiple servers means that part of building the game is deciding how to partition the load over these servers. Two techniques are commonly used in both online games and virtual worlds. Sometimes only one of the techniques is used, sometimes both, depending on the nature of the game or world.

The first technique is to exploit the geography of the game or world, decomposing the game into different areas, each of which can be mapped to a hosting server. For example, an island in Second Life corresponds to a physical server running the code for the shared reality of the world. Similarly, different areas of the World of Warcraft universe are hosted on different physical machines. Anyone who is in the area will connect to the same server, and interactions among the players on that server can be localized (and optimized). Actions happening in a different part of the world are not likely to affect those in this part of the world, so the communication traffic between servers can be kept small.

The second technique is known as *sharding*. A shard is a copy of a part of the game or virtual world. Different shards reside on different servers, and players who are assigned to one shard can interact with the world and other players in the shard, but will not see (or be able to interact with) players or objects in other shards. Shards not only allow more players to be supported in the world, but also permit independent explorations into the world by different sets of players. Thus, when a new quest or mission is added to a game, it will often be replicated with multiple shards so that more than one player (or group of players) can experience the quest or mission in its original state.

Although sharding and geographic decomposition allow multiple servers to be used to handle the load on a single game or world, they do present the developer with significant challenges. By creating noninteracting copies of parts of a world, shards isolate the players in different shards from each other. This means that players who want to share their experience of the world or game need to become aware of the different shards that are being offered, and arrange to be placed in the same shard. As the number of players who want to be in the same shard increases (some guilds—groups of players who cooperatively play in a single game over an extended period of time—have hundreds of members), the difficulty of coordinating placement into shards increases and interferes with the experience of the world. While shards allow scale, they do so at the price of player interaction.

Geographic decomposition does not limit player interaction, but does require that the designers of the game be able to predict the size of a geographic area that will be the correct unit of decomposition. If one geographic area becomes very popular, play on that area will slow down as the server associated with the area is overloaded. If a geographic area is less popular than originally predicted, computer hardware (and money) will be wasted on that section because not enough players are there. Since the geographic decomposition is hardwired into the code of the game or world, changing the decomposition in response to observed user behavior requires rewriting part of the game or world itself. This takes time, can introduce bugs, and is very costly. While this is being done, gameplay can be adversely affected. In extreme cases, this can have a

SCALING
in games & virtual worlds

major financial impact. When World of Warcraft was introduced, the demand for the game so outstripped its capacity that subscriptions had to be closed off for months while the code that distributed the game was rewritten.

CHANGING CHIP ARCHITECTURES

Scaling over a set of machines is a distributed computing problem, and the game and virtual-world programming culture has had little experience with this set of problems. This is hardly the only place where scaling requires the game programmer to learn a new set of skills. A change in the trend of chip design also means that these programmers must learn skills they have never had to exercise before.

With the possible exception of the highest end of scientific computing, no other kind of software has ridden the advances of Moore's law as aggressively as game or virtual-world programs. As chips have gotten faster, games and virtual worlds have become more realistic, more complex, and more immersive. Serious gameplayers invest in the very best equipment that they can obtain, and then use techniques such as overclocking to push even more performance out of those systems.

Now, however, chip designers have decided to exploit Moore's law in a different way. Rather than increasing the speed of a chip, they are adding multiple cores to a chip running at the same (or sometimes slower) clock speed. There are many good reasons for this, from simplified design to lower power consumption and heat production, but it means that the performance of a single program will not automatically increase when you run the program on a new chip. Overall performance of a group of programs may increase (since they can all run in parallel) but not the single program (unless it can be broken into multiple, cooperating threads). Games are written as single-threaded programs, however.

In fact, games and virtual worlds (and especially the server side of these programs) should be perfect vehicles to show the performance gains possible with multicore chips and groups of cooperating servers. Games and virtual worlds are embarrassingly parallel, in that most of what goes on in them is independent of the other things that are happening. Of the hundreds of thousands of players who are active in World of Warcraft at any one time, only a very small number will be interacting with any particular player. The same is true in Second Life and nearly all large-scale games or worlds.

The problem is that the culture that has grown up around games and virtual worlds is not one that understands or is overly familiar with the programming techniques that are required to exploit the parallelism inherent in these systems. These are people who grew up on a single (PC) machine, running a single thread. Asking them to master the intricacies of concurrent programming or distributed systems takes them away from their concentration on the game or world experience itself. Even when they have the desire, they don't have the time or the experience to exploit these new technologies.

PROJECT DARKSTAR

It is for these reasons that we started Project Darkstar (http://www.projectdarkstar.com), a research effort attempting to build a server-side infrastructure that will exploit the multithreaded, multicore chips being produced and scaled over a large group of machines, while presenting the programmer with the illusion that he or she is developing in a single-threaded, single-machine environment. Hiding threading and distribution is, in the general case, probably not a good idea (see http://research.sun.com/techrep/1994/abstract-29.html for a full argument). Game and world servers tend to follow a very restricted programming model, however, in which we believe we can hide both concurrency and distribution.

The model is a simple event-based one in which input from the client is received by the server, which then sets off a task in response to that event. These tasks can change the state of the world (by moving a player, changing the state of an object, or the like) and initiate communication. The communication can be to a single client or to a group of clients that are all subscribed to the same communication channel.

We chose this model largely because this is the way most game and virtual-world servers are already structured. The challenge was then to keep this model and allow servers written in this style to be scaled over multiple cores (running multiple threads) and multiple servers. We were not trying to take existing code and allow it to run within our system. This would have made the task much more difficult and would not have corresponded to the realities of the game and virtual-world culture. Game and world servers are written from scratch for each game or world, perhaps reusing some libraries but rarely, once running, being rehosted into a different environment. Efforts to bring different platforms into the game are

restricted to the client side, where new consoles bringing in new players may be worth the effort.

Darkstar provides a container in which the server runs. The container provides interfaces to a set of services that allow the game server to keep persistent state, establish connections with clients, and construct publish/subscribe channels with sets of clients. Multiple copies of the game server code can run in multiple instances of the Darkstar container. Each copy can be written as if it were the only one active (and, in fact, it may be the only one active for small-scale games or worlds). Each of the servers is structured as an event loop—the main loop listens on a session with a client that is established when the client logs in. When a message is delivered, the event loop is called. The loop can then decode the message and determine the game or world action that is the appropriate response. It then dispatches a task within the container.

Each of these tasks can read or change data in the world through the Darkstar data service, communicate with the client, or send messages to groups of other game or world participants via a channel. Under the covers, the task is wrapped in a transaction. The transaction is used to ensure that no conflicting concurrent access to the world data will occur. If a task tries to change data that is being changed by some other concurrent task, the data service will detect that conflict. In that case, one of the conflicting tasks will be aborted and rescheduled; the other task should run to completion. Thus, when the aborted task is retried, the conflict should have disappeared and the task should run to completion.

This mechanism for concurrency control does require that all tasks access all of their data through the Darkstar data service. This is a departure from the usual way of programming game or world servers, where data is kept in memory to decrease latency. By using results from the past 20 years of database research, we believe that we can keep the penalty for accessing through a data service small by caching data in intelligent ways. We also believe that by using the inherent parallelism in these games, we can increase the overall performance of the game as the number of players increases, even if there is a small penalty for individual data access. Our data store is not based on a standard SQL database since we don't need the full functionality such a database provides. What we need is something that gives us fast access to persistently stored objects that can be identified in simple ways. Our current implementation uses the Berkeley Database for this, although we have abstracted our access to it to provide the opportunity to use other persistence layers if required.

Concurrency control is not the only reason to require that all data be accessed through the data store. By backing the data in a persistent fashion rather than keeping it in main memory, we gain some inherent reliability that has not been exhibited by games or worlds in the past. Storing all of the data in memory means that a server crash can cause the loss of any change in the game or world since the last time the system was checkpointed. This can sometimes be hours of play, which can cause considerable consternation among the customers and expensive calls to the service lines. By keeping all data persistently, we believe we can ensure that no more than a few seconds of game or world interaction will be lost in the case of a server crash. In the best case, the players won't even notice such a crash, as the tasks that were on the server will be transferred to another server in a fashion that is transparent to the player.

The biggest payoff for requiring that all data be kept in the data store is that it helps to make the tasks that are generated by the response to events in the game portable. Since the data store can be accessed by any of a cluster of machines that are running the Darkstar stack and the game logic, there is no data that cannot be moved from machine to machine. We do the same with the communication mechanisms, ensuring that a session or channel that is connecting the game and some set of clients is abstracted through the Darkstar stack. This allows us to move the task using the session or channel to another machine without affecting the semantics of the task talking over the session or channel.

This task portability means we can dynamically balance the load on a set of machines running the game or virtual world. Rather than splitting the game up into regions or shards at compile time, virtual worlds or games based on the Darkstar stack can move load around the network of server machines at runtime. While the participant might see a short increase in latency during the move, the overall latency will be decreased after the move. By moving tasks, we not only can balance the load on the machines involved, but also try to collocate tasks that are accessing the same set of data or that are communicating with each other. All of these mechanisms allow us to determine, while the game is being played, which tasks (and which users) should be placed on the same server.

The project is in its early stages of development and deployment. It is based on an open-source licensing model and community, so we are relying on our users to educate us about the needs of the community that will build the games and worlds that will run on the infrastructure. The research is part computer science and part

anthropology, but each of the cultures has an opportunity to learn much from the other.

Even at this early stage, it is clear that this is going to be a complex venture. While early experience with the code has shown that the programming model does relieve the game or world server programmer from thinking about threads and locking, it has also shown that there are places where they do have to understand something about the underlying concurrency of the system. The most obvious of these is in the design of the data structures. One of the earliest users of our code was getting terrible performance from the system. When we looked at the code, we discovered that a single object was written to on every task, updating a global piece of game state. By designing the server in this way, this user effectively serialized all of the tasks that were running in the system, making it impossible for the server to get any advantage from the inherent parallelism in the game. Some minor redesign, breaking the single object into many (much smaller) objects, removed this particular bottleneck, with resulting gains in overall performance. This experience also taught us that we need to educate users of the system in the design of independent data structures that can be accessed in parallel.

Our own implementation has not been without some excitement. When we moved from a multithreaded server that ran on a single machine to an implementation that runs on multiple machines, we expected some degradation in the performance of the single-machine system. We were delighted to find that the single-node system degradation was not nearly as large as we thought it would be, but we found that additional machines lowered the capacity of the overall system. When presented with these measurements, this was not all that surprising to understand—the possibility for contention on multiple machines is greater than that on a single machine, and discovering and recovering from such contention takes longer. We are working on removing the choke points so that adding equipment actually adds capacity.

Measuring the performance of the system is made especially challenging by the lack of any clear notion of what the requirements of the target servers are. Game developers are notoriously secretive, and the notion of a characteristic load for a game or virtual world is not something that is well documented. We have some examples that have been written by the team or by people we know in the game world, but we cannot be sure that these are accurate reflections of what is being written by the industry. Our hope is that the open-source community that is beginning to form around the project will aid in the production of useful performance and stress tests.

Seen in a broader light, the project has been and continues to be an interesting experiment in building levels of abstraction for the world of multithreaded, distributed systems. The problems we are tackling are not new. Large Web-serving farms have many of the same problems with highly variable demand. Scientific grids have similar problems of scaling over multiple machines. Search grids have similar issues in dealing with large-scale environments solving embarrassingly, but not completely, parallel problems.

What makes online games and virtual worlds interestingly different are the very different requirements they bring to the table compared with these other domains. The interactive, low-latency environment is very different from grids, Web services, or search. The growth from the entertainment industry makes the engineering disciplines far different from those others, as well. Solving these problems in this new environment is challenging, and adds to our general knowledge of how to write software on the emerging class of multithreaded, multicore, distributed systems.

And best of all, it's fun. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

JIM WALDO is a Distinguished Engineer with Sun Microsystems Laboratories, where he conducts research on large-scale distributed systems. Prior to (re)joining Sun Labs, he was the lead architect for Jini, a distributed programming system based on Java. He spent eight years at Apollo Computer and Hewlett-Packard, where he led the design and development of the first object request broker and was instrumental in getting that technology incorporated into the first OMG CORBA specification. Waldo is an adjunct faculty member at Harvard University, where he teaches distributed computing in the department of computer science. He has a Ph.D. in philosophy, holds M.A. degrees in both linguistics and philosophy, and has never taken a real computer science course.

This article appeared in print in the August 2008 issue of Communications of the ACM.