TECHNICAL REPORT

Scalable Data Storage in Project Darkstar

Tim Blackman and Jim Waldo

> *Sun Microsystems Laboratories*

Blank page behind the cover

Scalable Data Storage in Project Darkstar

Tim Blackman
Jim Waldo

SMLI TR-2009-187 September 2009

Abstract:

We present a new scheme for building scalable data storage for Project Darkstar, an infrastructure for building online games and virtual worlds. The approach promises to provide data storage with horizontal scaling that is tailored to the special requirements of online environments and that takes advantage of modern multi-core architectures and high throughput networking.

After a brief overview of Project Darkstar, we describe the overall architecture for a caching data store. Then we provide more detail on the individual components used in the solution. Finally, we suggest some of the additional facilities that will be required to bring the full experiment to completion.

Sun Labs
16 Network Circle
Menlo Park, CA 94025

email addresses:
tim.blackman@sun.com
jim.waldo@sun.com

For information regarding the SML Technical Report Series, contact Mary Holzer or Nancy Snyder, Editors-in-Chief <Sun-Labs-techrep-request@sun.com>. All technical reports are available online on our website, http://research.sun.com/techrep/.

Scalable Data Storage in Project Darkstar

Tim Blackman, Jim Waldo
Sun Microsystems Laboratories
tim.blackman@sun.com, jim.waldo@sun.com

1 Introduction

Project Darkstar[1] is an investigation into massive scaling using modern multi-core architectures and distributed computing infrastructures. Unlike many other attempts to exploit these emerging computing paradigms, which tend to focus on embarrassingly parallel applications that are optimized for throughput, the target applications for the infrastructure being produced by this project are online games and virtual worlds. We chose these targets because of the particular set of requirements that must be met. First, such applications require massive scale, with thousands to hundreds of thousands of simultaneous users interacting across large numbers of servers. These applications deal with relatively small amounts of mutable data, and require very low latencies in the servers to preserve the interactive feeling of the applications. Finally, these applications require that a central facility, under control of the game administrator, be used to hold the true state of the game or world, to avoid problems of users cheating or otherwise "gaming" the system.

The Project Darkstar infrastructure attempts to present a simple programming model for the developers of the server-side of such games and virtual worlds. The goal is to allow these programmers to write their code as if that code were to be running in a single thread on a single machine. The code must be structured as a set of tasks triggered by events sent from the various clients of the game or world. The infrastructure will then take care of distributing those tasks over the available machines and threads, and running those tasks in such a way that the consistency of the data will not be compromised.

The scaling strategy used by the infrastructure centers on the properties of these tasks. The tasks are required to be compiled into Java™ bytecodes, and so the code for the tasks can be run on any machine in the server network without changing the semantics of that code. All of the tasks are run inside a transaction, allowing the underlying infrastructure to detect conflicts in attempts to access and manipulate shared data. When such a conflict is detected, one of the competing tasks will be allowed to continue, while any other task that was in conflict will be aborted and re-scheduled for running at a later time. To enable the infrastructure to detect such data conflicts, all of the data used by a task that might be visible to another task must be placed in a data store that is managed by the infrastructure itself.

By having the data store be a resource shared by all nodes running a game server, we also insure that all of the data within the store is available on all of these nodes. This means that we can move tasks from one machine to another to balance the load over the network. But this also means that the data store is a central component of the infrastructure itself.

2 Data storage in Project Darkstar

Like many other online applications, online games and virtual worlds access large quantities of persistent data, but with some important differences. Although there are many well known techniques[3][4][5][6][7][8] for implementing highly scalable databases to support typical online applications, the special characteristics of virtual environments make these standard approaches ineffective. The first difference is that, to support fast response times for users, the latency of data access is more important than throughput. Second, unlike most data-intensive applications, where data reads predominate, a higher proportion of data accesses in virtual environments involve data modification. While no full studies of access patterns have been widely publicized, our informal measures of typical applications indicate that modification of half of the data that is accessed is not unusual. Finally, unlike applications involving real world goods and payments, users are more willing to tolerate the loss of data or history due to a failure in the server, so long as these failures are infrequent, the amount of data lost is small, and the recovered state remains consistent. This reduced need for data durability provides an opportunity to explore new approaches to scalable persistence that can satisfy the needs of games and virtual worlds for low latency in the presence of frequent writes.

Achieving low latencies requires changes to the persistence implementation, even when running in a non-distributed environment. When storing data on a single node, the best way to provide low latency is to avoid the cost of flushing modifications to disk. Since online games and virtual worlds can tolerate an occasional lack of durability, a single-node system can continue with its processing without waiting to force data modifications to disk, allowing the modifications to be flushed in the background. This behavior is acceptable so long as the system can preserve integrity in the case that some modifications have not reached the disk at the point of a node failure. Moving disk flushes out of the critical path in this way allows the system to take full advantage of disk throughput. In our tests, a database transaction that modifies a single data item takes more than 10 milliseconds if the operation includes performing a disk flush, but as little as 25 microseconds without flushing.

Long network latencies pose a similar problem for data storage in multi-node systems. As network speeds have increased, network throughput has improved dramatically, but latency continues to be substantial. In simple tests of network performance running over an 8 Gigabit Infiniband network, and using the best software we could find for use in a Java environment (standard sockets in a prerelease version of Java™ Platform, Standard Edition 7 with Sockets Direct Protocol (SDP) to perform TCP/IP operations over Infiniband), the best result we were able to achieve was a network round trip latency of 40 microseconds. Adding this additional 40 microseconds to the 25 microsecond transaction time possible on a single node threatens to reduce performance significantly.

Reducing latency is not the only problem that needs to be addressed to produce a good scaling solution. To support horizontal scaling, increasing system capacity should be as simple as adding additional application nodes as needed. Adding nodes in this way is only possible if the newly added nodes are stateless — they should not store vital data that needs to be maintained to insure the system's integrity. If an application node fails, it should be possible to replace it with a new node without needing to recover data from the failed one. Users that were connected to the failed node will need to reconnect to a new node, and may find that some modest amount of their recent in-world history has been rolled back, but should otherwise be able to proceed with their experience.

While individual application nodes should not maintain definitive copies of any data, all nodes still need to be able to access all of the information in the game. This free access to all game data is needed to

allow developers to write their game logic as if running on a single machine, rather than needing to explicitly partition the game into separate regions running on different nodes.

The first approach taken to implementing such a scalable data storage scheme was for nodes to request and store data using a central data server, without caching any data locally on the individual nodes. This scheme was easy to implement, and was introduced a couple of months after the first release of Project Darkstar's current architecture. Unfortunately, the need for a network round trip for each operation in this version resulted in poor performance due to network latency. Faster networks with improved throughput would not change these results significantly.

The original plan had been to improve the scalability of the no-caching scheme by implementing support for multiple data servers. If a central data server had provided good performance, then switching to multiple data servers would have added the needed scalability. The prohibitive cost of network latency, though, made this approach unworkable. A further complication would have been the need to migrate data explicitly among the multiple data servers, which would have introduced further complexity.

Another idea had been to store data on each application node. That idea had the drawback of requiring backup and failover redundancy for each application node, which would have clashed with the desired ability to add and remove applications nodes on demand. This scheme, too, would have required explicit migration of data, this time among application nodes.

This document describes a new approach: using write caching with a central data server. The idea is to cache data locally on each node, including modified data, so long as it is only being used by that node. If local modifications need to be made visible to another application node because the node wants to access the modified data, then all local changes need to be flushed back to the central server, to insure consistency.

This new scheme avoids network latency so long as the system can arrange for transactions that modify a particular piece of data to be performed on the same node. It avoids the need for explicit object migration: objects will be cached on demand by the local node. It also permits adding and removing application nodes, and avoids the need for redundancy and backup, since application nodes do not store globally important data.

3 Architecture

The overall architecture is similar to a standard client/server caching scheme with callbacks, but with some changes to support caching modifications. Each application node has its own *Data Cache*, which maintains local copies of recently used items. Data caches communicate with the central *Data Server*, which maintains persistent storage for items. The data server also keeps track of which items are stored in which data caches, and makes callback requests to those caches to request the return of items that are needed elsewhere. We assume that the central data server and all of the application nodes on which data caches are located are mutually trusting, and that there are dependable network links among the nodes. In the common case, the central data server and the nodes on which the caches reside will be co-located in a single data center, but this is not required for the design.

3.1 Data Cache

When an application node asks the data cache for access to a data item, the cache first checks to see if the item is present. If the item is present and is not being used in a conflicting mode (write access by one local transaction blocks all other local access), then the cache provides the item to the application immediately. If a conflicting access is being made by another transaction, the access request is queued in the data cache's lock manager and blocks until all current transactions with conflicting access, as well

as any other conflicting accesses that appear earlier in the queue, have completed.

If the item is not present in the cache, or if write access is needed but the item is only cached for read, then the data cache contacts the data server to request the desired access. The request either returns the requested access, or else throws an exception if a timeout is detected. If additional transactions request an item for reading from the cache while an earlier read request to the server is pending, the additional access waits for the results of the original request, issuing an additional request as needed if the first one fails.

Because data stored in the data cache can be used by multiple transactions on the application node, requests to the data server are not made on behalf of a particular transaction. The lack of a direct connection between transactions and requests means there needs to be a way determine the proper timeout for a request. One possibility would be to provide a specific timeout for each request, based on the time remaining in the transaction that initiated the request. Another approach would be to use a fixed timeout, similar to timeout that the system applies to each transaction, to better model the fact that a request may be shared by multiple transactions. A fixed timeout increases the chance that a request will succeed so that its results can be used by other transactions on the application node, even if the transaction initiating the request has timed out. For this reason, we have taken this approach, and the system has a fixed timeout for all calls to the server.

When a transaction commits, the commit updates the data cache with the new values, insuring that the cache updates appear atomically. The changes are then stored in the *Change Queue*, which forwards the updates, in order, to the data server. Ordering the updates by transaction insures that the persistent state of the data managed by the data server represents a view of the data as seen by the application node at some point in time. The system does not guarantee that all modifications will be made durable, but it does guarantee that any durable state will be consistent with the state of the system as seen at some earlier point in time. Since transactions commit locally before any associated modifications are made persistent on the server, application developers need to be aware of the fact that a failure of an application node can result in committed modifications made by that node being rolled back.

If an item needs to be evicted from the data cache, either to make space for new items or in response to a callback request from the data server, the data cache waits to remove the item until any modifications made by transactions that accessed that item have been sent to the data server. This requirement insures the integrity of transactions by making sure that the node does not release locks on any transactional data until the transaction has completed by storing its modifications.

Strictly speaking, the change queue would not need to send changes to the data server immediately in order to maintain integrity, so long as changes were sent before an item was evicted from the cache. There is no obvious way to predict when an eviction will be requested, though, and the speed of eviction will affect the time needed to migrate data among application nodes. To reduce the time needed for eviction, the best strategy is probably to send changes to the data server as the changes become available. The node need not wait for the server to acknowledge the updates, but it should make sure that the backlog of changes waiting to be sent does not get too large. This strategy takes advantage of the large network throughput typically available without placing requirements on latency, a key advantage over the no-caching scheme. There are various possibilities for optimizations, including reordering unrelated updates, coalescing small updates, and eliminating redundant updates.

The data cache provides a *Callback Server* to handle callback requests from the data server for items in the cache. If an item is not in use by any current transactions, and was not used by any transactions whose changes have not been flushed to the server,

then the cache removes the item, or write access to the item if the request is for a downgrade from write to read access, and responds affirmatively. Otherwise, the callback server responds negatively. If the item is in use, the callback server queues a request to access the cached item. When access is granted, or if the item was not in use, the callback server queues the callback acknowledgement to the change queue. Once the acknowledgement has been successfully sent to the data server, then the change queue arranges to remove the access from the cache.

The data cache assigns a monotonically increasing identifier to transactions that access data. Items that were used during a transaction are marked with the ID of the transaction with the highest ID for which they were used. This ID is used to determine when an item can be evicted in response to a callback request, as well as for the algorithm the cache uses to select old items for eviction.

The data cache needs a way to determine when it is full and should evict old items. The simplest approach is to specify a fixed number of entries as a way of limiting the amount of data held in the cache. Another approach would be to include the size of the item cached in the estimate of cache space used. A still more complicated approach would involve making an estimate of the actual number of bytes consumed by the data structures needed to store a particular cache entry, and computing the amount of memory available as a proportion of the total memory limit for the virtual machine. We are currently using the first approach, as we have found that the added information of the other approaches does not make up for the additional complexity.

Our previous experience with Berkeley DB, reinforced by published research[2], suggests that the data cache should perform deadlock detection whenever a blocking access occurs, and should choose either the youngest transaction or the one holding the fewest locks when selecting the transaction to abort. Since the previous research found the two approaches comparable, we have adopted the approach of aborting the youngest transaction, again because of the simplicity of the code needed to implement that approach.

3.2 Data Server

The central data server maintains information about which items are cached in the various data caches, and whether they are cached for read or write. Nodes that need access to items that are not available in their cache send requests to the data server to obtain the items or to upgrade access. If a requested item has not been provided to any data caches, or if the item is only cached for read and has been requested for read, then the data server obtains the item from the underlying persistence mechanism, makes a note of the new access, and returns it to the caller.

If there are conflicting accesses to the item in other data caches, the data server makes requests to each of those caches in turn to call back the conflicting access. If all those requests succeed, then the server returns the result to the requesting node immediately. If any of the requests are denied, then the server arranges to wait for notifications from the various data caches that access has been relinquished, or throws an exception if the request is not satisfied within the required timeout.

When the data server supplies an item to a data cache, it might be useful for the server to specify whether conflicting requests for that item by other data caches are already queued. In that case, the data cache receiving the item could queue a request to flush the item from the cache after the requesting transaction was complete. This scheme would probably improve performance for highly contended items.

3.3 Networking and Locking

The data caches and the data server communicate with each other over the network, with the communication at least initially implemented using Java™ Remote Method Invocation (Java RMI), for

simplicity. In the future, it might be possible to improve performance by replacing Java RMI with a simpler facility based directly on sockets. For the data cache's callback queue, it might also be possible to improve performance by pipelining requests and using asynchronous acknowledgements.

Both the data cache and the data server have a common need to implement locking, with support for shared and exclusive locks, upgrading and downgrading locks, and blocking. The implementation of these facilities is shared as much as possible. Note that, because the server does not have information about transactions, there is no way for it to check for deadlocks.

4 Localizing data access

The caching scheme described here will work best when access to a particular set of data items is localized on a single node. In such a case, the data cache on that node can be used by all local tasks that access those items, with only the need for asynchronous writes to the central data server. The worst case for this scheme is when all data is being accessed for write on multiple nodes. Such an access pattern will result in pathological behavior leading to poor performance where the data being accessed on multiple nodes will need to be evicted from local caches repeatedly. In this case, the performance of the cache would revert to the behavior of the non-caching implementation, but with additional overhead to maintain the cache.

We are currently experimenting with mechanisms that will allow for the proper clustering of data access on a single node. To get the kind of performance needed, the clustering does not have to be perfect, but merely sufficient to avoid the pathological cases.

References

[1] Project Darkstar. http://www.projectdarkstar.com/

[2] R. Agrawal, M.J. Carey, and L.W. McVoy. December 1987. “The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems”, *IEEE Transactions on Software Engineering 13,* no. 12: 1348-1363.

[3] Memcached. http://www.danga.com/memcached/

[4] Jboss Cache. http://www.jboss.org/jbosscache/

[5] Wikipedia. Partition (database). http://en.wikipedia.org/wiki/Partition_%28database%29

[6] B. Nitzberg and V. Lo. 1991. “Distributed Shared Memory: A Survey of Issues and Algorithms”, IEEE *Computer:* 24, issue 8: 52-60.

[7] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. 1991. “The ObjectStore Database System”, *Communications of the ACM*: 34, no. 10: 50-63.

[8] Oracle. Oracle Berkeley DB: Replication. http://www.oracle.com/technology/products/berkeley-db/db/index.html

About the Authors

Tim Blackman is a Staff Engineer for Sun Microsystems Laboratories, working on Project Darkstar. His research interests include Databases, Distributed Systems and Java. Prior to joining Sun Labs, he was a member of the Sun Jini™ Technology Group, working primarily on security and configuration facilities for the Jini 2.0 release. Before joining Sun, he worked on object-oriented databases and electronic CAD tools.

Jim Waldo is a Distinguished Engineer with Sun Microsystems, where he is the technical lead of the Darkstar project. Prior to (re)joining Sun Labs, Jim was the lead architect for Jini, a distributed programming system based on Java. While at Sun, Jim has done research and product development in the areas of object-oriented programming and systems, distributed computing, and user environments. Before joining Sun, Jim spent eight years at Apollo Computer and Hewlett Packard working in the areas of distributed object systems, user interfaces, class libraries, text and internationalization. While at HP, 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. He edited the book *The Evolution of C++: Language Design in the Marketplace of Ideas* (MIT Press), and was one of the authors of *The Jini Specification* (Addison Wesley).

Sun Microsystems Laboratories
16 Network Circle
Menlo Park, CA 94025

Scalable Data Storage in Project Darkstar

Tim Blackman and Jim Waldo

SMLI TR-2009-187