

WHITE PAPER

ADVANCED DATA MANAGEMENT FOR MMOG

By Robert Greene, Vice President Open Source Operations,
Versant Corp.

INTRODUCTION

MMOG's (massive multi-player online games) fall firmly into the category of the most demanding applications in the world. MMOG requirements such as: real-time response, millions of users, thousands of concurrent connections and increasingly complex simulation models intertwined with increasingly unique billing requirements all coupled with growing amounts of data requiring high availability, reinforce the reality of difficulty in delivering such an application to the gaming community.

The unique requirements of MMOG to the gaming community involve the necessity to deliver these capabilities into a server based platform. The server platform provides among other things persistence of state in the game such that when a gamer leaves, the game continues and game state continues to change. Clearly, the continually changing state requires that game state be permanently stored and retrievable.

This is in direct contrast to the traditional game which always starts from the same repeating point and this unique requirement mandates the integration of a highly scalable data tier unlike that seen in any other gaming platform. The following will explore the challenges in delivering an MMOG solution for the data layer and describe how the Versant Object Database delivers on the capabilities required by these incredibly demanding applications.

The Versant Object Database comes with the following architecture, everything below the network communications routing, "out of the box" and all discussions will reference the basic capabilities found in this diagram.

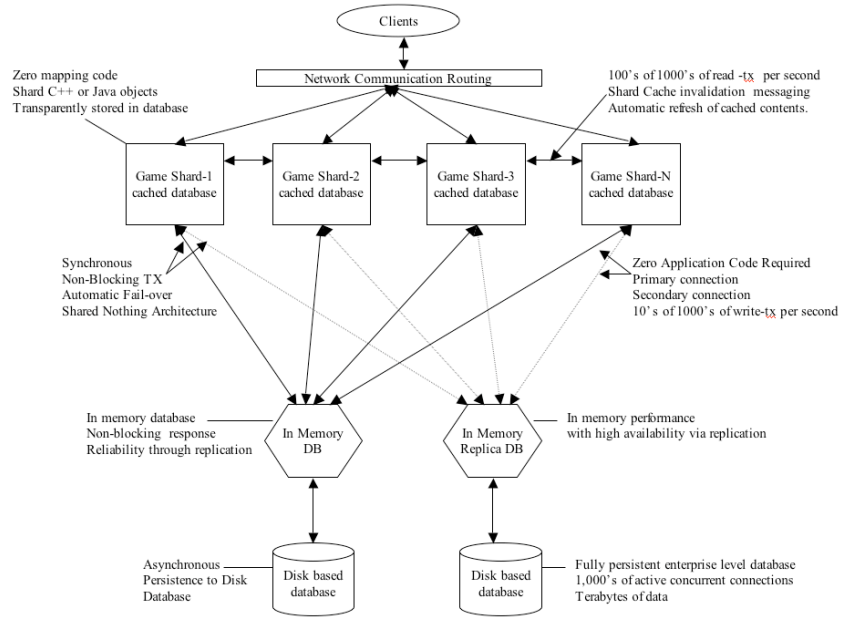


Fig. 1 - Architecture Versant Object Database in MMOG Applications

40%

less code when using native persistence

- » No separate data definition language
- » No mapping code

ADVANTAGES FOR CORE MMOG REQUIREMENTS

The Versant Object Database provides native persistence for C++, Java and C# object models[1].

Native persistence means (1) there is no separate data definition language to describe the storage structure of game objects in the database and (2) there is no mapping code required to move the 'data' from your database into your game model objects. In traditional relational databases, 'data' is described by a DDL (data definition language) and held in a different structure and type system in the database than the in memory language model. As a result of this mismatch, significant code is required to be developed and executed at runtime to bring the 'data' into your game models.

Using native persistence results in the elimination of what is often as much as 40% of the games manageable code base. The only database related code you need to concern yourself with is getting

No

mapping
means

- » Reduced time to market
- » Reduced deployment footprint
- » Substantial performance improvements

connections, getting initial use case game objects (either directly using an id or via a server side query execution that returns game objects), and where to call commit to flush game changes to the database. All other behavior involved in writing a persistent system, such as dirty object tracking, new object creation, object deletion, shard cache management [2], are handled by Versant.

Mapping code elimination has three profound affects. (1) significantly reduced time to market, due to less code having to be written, maintained, debugged. (2) significantly reduced deployment footprint of the game, which reduces the TCO (total cost of ownership). The reduced footprint is due to the fact that significantly less CPU cycles are required in your game shards since the mapping code does not need to get executed. (3) substantial performance improvements, due to less code execution and lower CPU cycles in the server processes which use navigational access instead of relational queries.

Versant has done studies with application server vendors illustrating the substantial savings in CPU for the shard tier. An IBM Redbook

<http://publib-b.boulder.ibm.com/abstracts/sg246561.html?Open>)

was created around a use case for a simple financial application where Versant showed a 75% reduction in CPU's for the application server tier while improving performance through the data tier by 50X. The study was done using an earlier version of the Versant database known as enJin, a Versant integration for application servers.

EVOLUTIONARY GAME MODELS

Versant also provides advantages in the area of flexibility in evolving the existing database schema and in creating dynamic game object characteristics.

This flexibility is achieved through support of both easier design and model extensibility. Versant deals with object oriented models in the proper polymorphic forms of the language and treats

references to related objects orthogonal to the state of the objects involved.

This means that you can write game logic to deal with interfaces or abstract classes that are then further derived to create the specialized classes for your game without introducing needlessly complex application and mapping code. Performance optimizations created at the abstract level allow you to add new game classes while keeping your performance optimizations intact. The original game logic incorporates new game object use with minimal code modification. Game object attributes can be easily managed through objectified relationships without introducing complicated coding semantics.

A simple design example could be the modeling of a character Backpack. One obvious design when properly using OO would be to model the Backpack as an Item that contains other Items. The design uses both inheritance and self referencibility. This design is handled seamlessly using Versant and retrieval of a Backpack's contained Items requires nothing more than a message send in the game logic code. This same design in a relational form, even if flattening of the Backpack into a single Item table, forces performance draining JOIN operations due to the self referencibility.

A more complex extensibility example could be the creation of an entirely new game character by subclassing an existing class that describes general character behavior. The addition of the new character class can be deployed to the database server on the fly. The Versant Object Database will incorporate the new schema and evolve the database automatically, and existing game logic will continue to operate uninterrupted bringing the new character into the game with minimal effort.

SCALABILITY

When designing an MMOG software system for scalability, one should consider the ability to scale both vertically and horizontally.

Vertical scalability means that a single hardware node in the software system can maintain a set capacity of operations under required performance criteria by adding resources to the hardware node.

Horizontal scalability means that when reaching the tolerable threshold of vertical scalability, continued capacity of operations expansion can be achieved by adding additional hardware nodes. Maximizing Vertical scalability is important as this generally has a direct correlation to profit margins as a given configuration of hardware and software infrastructure supports a fixed number of users which translates into revenue. Additionally, maximizing vertical scalability reduces the complexity and overhead of maintainability and further improves margins.

Ensuring horizontal scalability over vertical scalability means that unlimited expansion is possible with predictable margins. Versant enables developers of MMOG applications to implement both vertical and horizontal scalability through an advanced multi-threaded architecture integrated with a number of specific features described below.

REAL-TIME RESPONSE

A significant advantage of Versant is the ability to cache all or portions of the database in the game shard memory space and/or proxy data server for real-time read response [3].

This shard caching is critical for meeting the minimized latency requirements of an MMOG and significantly improves vertical scalability by off-loading activity from the database server processes relieving contention for shared resources.

The "native" persistence capabilities also provide transparency of access. For example, if a character game object is loaded into the shard, either by id or query, then in the game logic a message is sent to the charter to retrieve a wallet for money, if the wallet did not exist in the shard memory space, Versant will transparently retrieve that wallet from the database server memory space

without the application executing any special query code for the wallet retrieval.

Game objects are implicitly assigned system global id's so that they can be retrieved without impacting application logic. There are also capabilities in Versant so when a game object is retrieved from the server memory space, a configurable number of related game objects will automatically be retrieved into the shard to guarantee a subsequent call to the database server memory is not required, thereby avoiding unnecessary network calls.

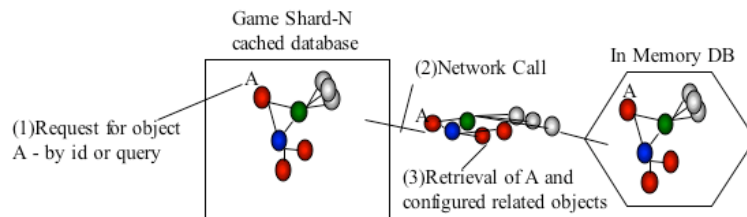


Fig. 2 – Real-time Response

DISTRIBUTED STATE MANAGEMENT

Versant has another feature working in concert with the real-time shard cache that facilitates distributed state management across multiple shards [4].

Shared game objects in the shard memory space can be identified so that changes made to the shared game objects cause cache invalidation notifications to be broadcast to all shards. All shards in the cluster that have those shared objects will then automatically refresh the shared objects from the database server memory to ensure a consistent and up to date game state across the shard cluster. This allows unprecedented horizontal scalability for large numbers of users with controlled cost structure by allowing incremental addition of shards as the player demand and concurrency builds. Further, network communications routing can use load balancing to take advantage of this clustering by reassigning incoming requests for better response or failover without incurring significant performance penalties.

In addition, this allows the design of a game where players and related game objects can effectively "walk" shard boundaries for uninhibited game play.

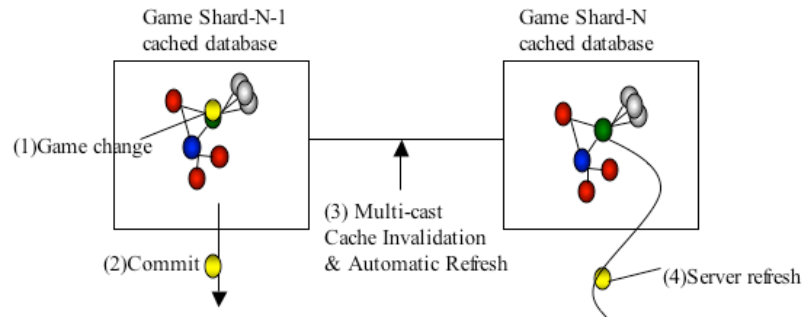


Fig. 3 – Distributed State Management

TRANSACTION THROUGHPUT

Versant has the unique ability to provide in-memory performance characteristics while maintaining the ACID (Atomicity, Consistency, Isolation, Durability) properties of an enterprise level database management system.

Versant makes this possible by leveraging synchronous in memory databases coupled with the ability to recover in-flight transaction failures. Versant uses asynchronous socket communications to provide highly efficient 2-phase commit protocol across distributed in-memory databases. Further, commit calls to all databases are non-blocking allowing for immediate return to application control with only the overhead of a network call. Contents of commit operations are stored in memory based transaction log buffers which are asynchronously flushed to the disk based logging subsystem by background threads every N (configurable) number of commits. Versant maintains durability for transactions by ensuring that transactional content resides synchronously in multiple buffer spaces in a shared nothing architecture [5]. The rapid database server response, due to this architecture, facilitates unprecedented vertical scalability by minimizing contention for internal structures in the server processes and eliminating waiting due to I/O operations.

The result is a database node that can support thousands of concurrent update transactions.

The game shard manager tracks all CRUD (create, read, update, delete) operations in the game logic. When the game logic calls commit, Versant uses asynchronous socket calls to get the changes (results of the transactional CRUD operations) to flow towards both in memory databases. The transactional changes are written into piggy backed double buffers . Piggy backed double buffers work in a way that a buffer is always available for writing even when one side of the buffer is being flushed to disk. Once the server threads have written into the piggy back buffers, control is immediately returned to the shard, so that real-time processing can continue. When the configured buffer flush threshold is reached, the piggy back buffers reverse, new transactions are written into the opposite side and an asynchronous thread flushes changes to disk. This process continues during normal operations. If any failure should occur, Versant recovers on the fly as described in the section below on high availability.

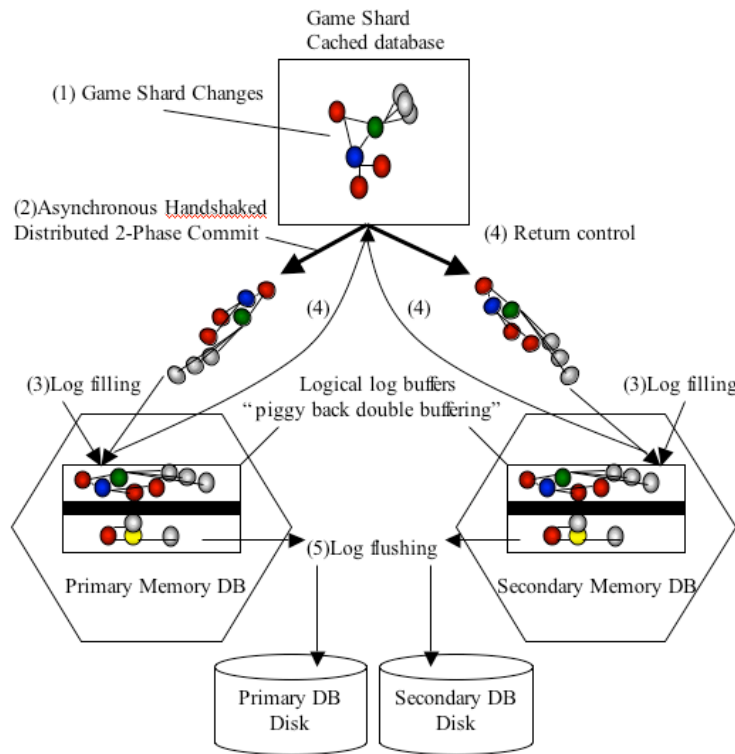


Fig. 4 – Transaction Throughput

PARALLEL PROCESSING

Versant provides the ability to incrementally add physical databases to a cluster which are treated as parallel processing nodes in a distributed system.

Operations performed on the nodes are optimized using threading and asynchronous communications to provide parallel operations for optimal performance. For example, the Game logic can define many logical groupings of physical databases and treat them each as a virtual datastore.

SQL92 type indexed queries can be targeted at a logical database and are executed in parallel within the physical in-memory databases representing the virtual datastore. By controlling the size of any physical database node within the cluster, the game logic can guarantee response times by controlling concurrent access and internal structure traversal times. Unlike in traditional relational databases where queries target a table and its columns, returning records that need to be mapped into your language objects, the Versant database queries target a class and its attributes, returning objects of that class without any intermediate application mapping.

Once the queried objects are retrieved into the game logic, access to related objects occurs through the normal message send process of the language. An example is shown in the following figure. A query can be performed across many in-memory databases in parallel to find a particular player. Once the player object is returned by the query, the game logic sends a message to the player to get the related backpack. There can be specialized messages defined so that whenever a backpack is retrieved it's contained items are also automatically retrieved.

The message `getBackpack` is sent to the player and the Versant database will ensure that the related backpack and it's contained items are retrieved into the shard cache. The Versant database handles all of the issues dealing with locality of the related objects in the physical cluster. There is no required code in the game logic to deal with determining where objects are physically located. Despite the fact that the backpack and its items are not

in the same physical database as the player, they are still transparently returned on message send to the shard cache. There is no limit to the type of architectural flexibility gained with this design, allowing any type of partitioning scheme deemed appropriate for your game design. Versant guarantees uniqueness of object identity across the entire cluster and maintains the physical locality orthogonal to the logical identity. Even if external processes are put in place to move objects from one physical database to another, there will be no impact to gaming logic. Versant even handles the logical and physical consistency of backups across the distributed cluster.

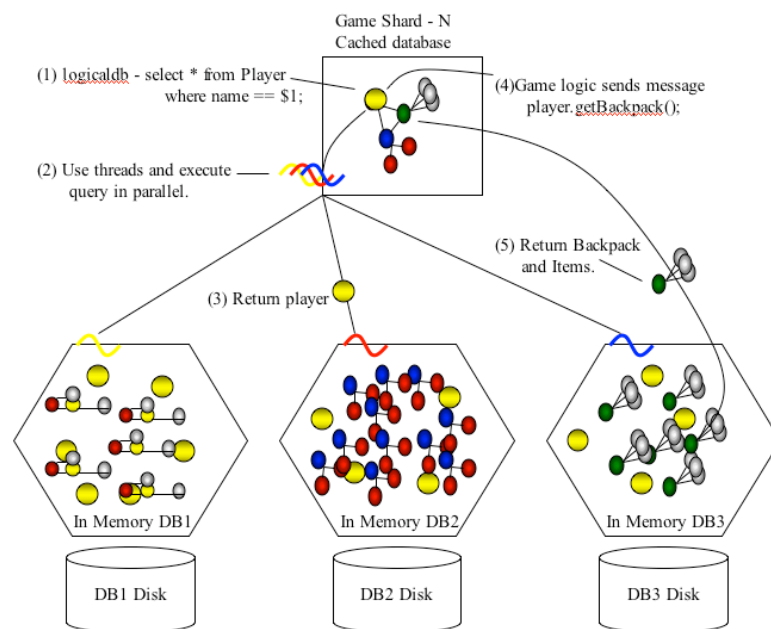


Fig 5 – Parallel Processing

DATA CLUSTERING

Another scalability issue for relational databases is disk access optimizations when dealing with data sizes that cannot be kept completely in memory.

Due to the unique ability of Versant to manage objects by logical identity, cluster schemes can be created to minimize disk access on retrieving related game objects.

Traditional relational databases have a difficult time optimizing access to disk because of the way both indexes and tables play such a critical role in data availability.

As shown in the illustration below, accessing a set of related objects that are stored in relational table structures will take multiple disk seeks (the most expensive I/O operation) to retrieve the related data into memory. There will potentially be several disk seeks for relevant index segments, then more disk seeks for each table space that is not in memory. If JOIN's are involved there may be lots of disk seeks to perform the cross product multiplication involved in the relational algebra operations. Relational database implementations attempt to work around these performance issues with expensive RAID disk striping systems.

Using RAID, data is spread across dedicated disk spindle segments mimicking parallel access to minimize delays. These RAID systems can help by squeezing multiple disk seeks into the same time slot, but RAID is very expensive, adding to the total cost of ownership in a relational based solution. Plus, RAID can also be used with the Versant database.

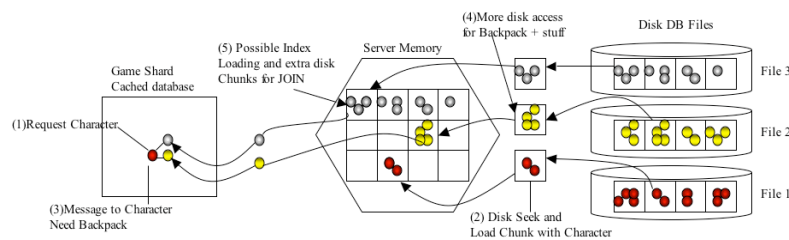


Fig 6 – Relational Database Data Load

Versant avoids the overhead in disk seek activity by physically clustering logically related game objects.

Let's assume a character comes into play that is not currently loaded in the shard or in memory database. This means that when loaded, a disk seek must be done to retrieve the character object into the in memory db and return it to the shard.

All databases manage data on disk by paging chunks of disk into memory, so the retrieval of the character means a chunk was loaded. By using physical object clustering, other objects logically related to the character, like it's backpack, team members,

activity history, etc will be physically clustered on disk. So, when the disk seek is done to move the chunk of disk containing the character into memory, Versant has additionally loaded all the logically related objects. As a result, when the game logic works with the character's logically related objects, they will already be loaded into the memory database and no further disk seeks will be required.

The performance advantage of an in memory lookup compared to a disk seek is enormous and further the immediate return of that lookup serves to decrease overall contention in the database server processes causing a dramatic overall improvement in throughput under concurrency. Physical Versant database sizes can be incrementally scaled by dynamically adding files or raw partitions to the database store. The Versant database has proven deployments in the multi-terabyte range.

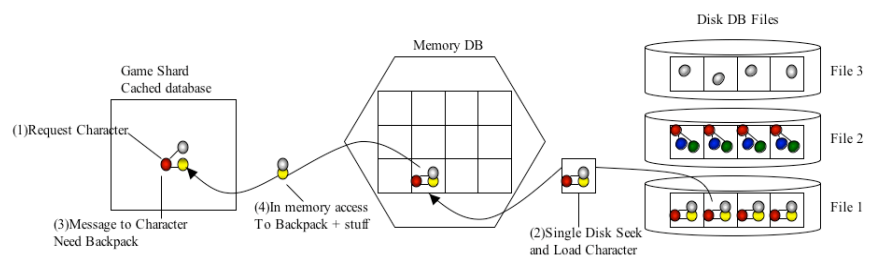


Fig. 7 – Versant's Physical Clustering

HIGH AVAILABILITY

Using Versant, if a failure occurs in a database server process due to network, hardware, software process failure, the shard will transparently receive the failure notification and change its mode of operation to continue the failed transaction only to the remaining live server [6]. Versant handles in-flight failures from either primary or secondary site transparently.

During failover operation, the shard sends additional transaction information to the live server to be used for automatic background restoration of the failed site. The live site forks a polling process that seeks to start the failed database server. When the failed site is reachable, the polling process uses the change set records to automatically resynchronize the failed site with the live site

production database server. Once synchronized, the polling process sets to server to notify the shard to begin synchronous operations. The Recovery of the failed site requires no DBA intervention, unless of course something catastrophic at the hardware level has occurred like complete disk failure. If desired, during the failover operation period, the live server can be configured to automatically change modes to provide synchronous buffer flush on commit[5].

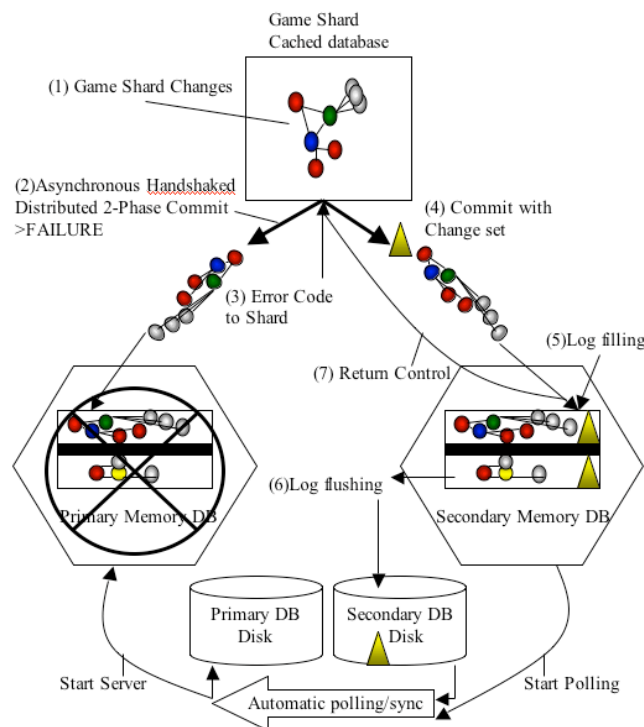


Fig. 8 – High Availability

MAINTAINABILITY

Versant provides enterprise class maintenance capabilities. DBA activities are significantly reduced because the internal architecture is simplified for object based storage.

This internal simplification means that traditional DBA maintenance issues like backup, indexing, buffer management, query execution optimizations, etc are minimized. All maintenance features

of the Versant database are online requiring zero downtime of the server processes. Versant provides integration with high end disk manufacturers like EMC so that online database backups of terabytes of data can be done in seconds including logically and physically consistent backup across distributed databases.

Versant provides SNMP agent based monitoring capabilities to monitor multiple physical databases in a centralized visual tool or integrated into your favorite network management suite. Other Versant tools provide capabilities such as data localization through master-slave or peer-to-peer replication, online reorganization, object inspection, etc.

CONCLUSIONS

The MMOG application space demands a real-time, high concurrency, large scale, enterprise class database solution. These combined requirements do not fit well with traditional relational technology.

The Versant database provides a combination of features specifically designed for the real-time, high scalability requirements of the MMOG application space. Versant's core database engine known as The Versant Object Database has been in commercial deployment for over 15 years and is a proven application specific database technology used in the most demanding applications in the world. The Versant database server is being used by: The American Stock Exchange, Sabre Online Reservations, The Financial Times content servers, GE Transportation Railway management systems, PeopleSoft supply chain management, Verizon Fraud Detection, Lockheed - War Games 2000, Boeing - LIDS's -National Missile Defense Simulations, Exxon Mobile Reservoir Simulations, Gundam Online and a very long list of other equally impressive large scale applications.

Reduce your risk, lower your costs, improve your performance and scalability while lowering your total cost of ownership.



Robert Greene

Robert Greene is Vice President Open Source Operations at Versant. This includes working with the db4o open source community.

Prior to his work with Versant, Robert Greene was Chief Engineer and Director of Operations at Seaboard Systems.

TECHNICAL NOTES

[1] Versant offers several APIs for different language implementations including C++, Java and C. The C++ implementations include: a proprietary API and standards based Object Database Management Group (ODMG) API. The Java implementations include: a proprietary API (JVI - Java Versant Interface) and standards based Java Data Objects (JDO) API. The C API is a fundamental access API which is accessible through C or through extensions into JVI and C++. For the .NET environment, Versant will introduce in 2008 an API which will allow to use and mix C#, J# and VB .NET inside the same application, using the same database.

[2] A shard cache is represented in the Versant system by a number of different implementations depending on your chosen language and API. For example, it is known as a "VSession" in C++, a FundSession or TransSession in JVI, a PersistenceManager in JDO. Regardless, those cache implementations behave in essentially the same manner. The cache can hold objects resident in your application memory space across transaction boundaries with varying degrees of lock and schema control. You can additionally take some control through both external configuration and API to hold and release objects in/from the cache as necessary for your application logic.

[3] Use of Versant in the architecture of an MMOG can come in varied forms depending on the needs of the game and it's targeted scalability. The Versant client, with its essentially a cache and connection to the underlying database storage and query engine, can be employed in any application tier that makes sense for your design. You may work with it directly integrated as your shard cache or you may design so that multiple shards are sharing an underlying cache (or proxy data server). These choices are often influenced by the nature of your game. For example, are you trying to create a "one world" game where everyone is essentially playing together or are you segmenting the population into smaller sub-units. These higher level goals will influence how you use Versant in your MMOG architecture.

[4] The facilities for distributed cache state management vary between language bindings. When using Versant's Java JDO API, the multi-cast facilities are part of the product and come in the form of an implementation on top of JGroups. When using JVI or C++, Versant consulting can assist in providing guidance on how to integrate any 3rd party multi-casting libraries with which you are already familiar or provide guidance in selecting a provider. Consulting examples for using JGroups with JVI and Spread for C++ are available on request.

[5] Versant uses a configuration option to enable asynchronous I/O which effectively turns Versant into an in-memory database by not blocking client requests on transaction boundaries for I/O operations. When using this feature, you are at risk of losing some transactions if the system were to crash. To mitigate that risk, Versant provides a solution known as the Fault Tolerant Server (FTS). When using FTS in conjunction with asynchronous I/O, the risk of losing any transactions is significantly reduced because a loss can only occur if there is a double failure of FTS. A double failure is an extremely rare event, as the FTS solution is providing database server availability that exceeds 5-9's up time requirements (.999999 availability). A future version will provide the configuration option to 'revert' to synchronous commit during failover operation.

[6] The shard here is presumed to be implemented as an integration with the Versant client (cache). The use of FTS then enables transparent failover if there is a loss in connectivity to the database server process. If you are implementing your shards over a proxy data server which uses the Versant cache, then the failover capabilities are moved to the proxy's tier.