White Paper

GemStone Facets | Delivering Scalable, Transactional Java Applications

The next generation of Java™ business systems requires the ability for thousands of users to act as if they were speaking one to one, to interact with a world of information as though it were all at their fingertips, to see myriad applications as one continuous business process. The next great gains will be achieved through systems that use near real-time performance to speed and scale complex transactions beyond the limitations of conventional Java and Java 2 Enterprise Edition™ (J2EE) architectures. This paper shows how GemStone Facets™ provides a proven, powerful foundation for these strategic initiatives.

Table of Contents

What Are the Technical Challenges? 3
Avoiding Data Conversion 3
Managing Large Oject Graphs in the JVM 3
Avoiding Redundant Reading of Persistent Objects 3
Sharing Live Information across Virtual and Physical Machines 4
Reducing Time in Transactions 4

GemStone Facets: Real-world Technology for Real-time Results 4
Architecture Of A Real-Time Engine 5

How Does GemStone Facets Help? 7
Avoiding Data Conversion 7
Managing Large Object Graphs in the JVM 8
Avoiding Redundant Reading of Persistent Objects 9
Sharing Live Information across Virtual and Physical Machines 10
Reducing Time in Transactions 11

Integration into J2EE Applications 12
Accessing Objects in J2EE Applications 12
Controlling Transactions in J2EE Applications 13

Conclusions 13

Application Usage Scenerios 15
GemStone Facets as a Cache 15
GemStone Facets as a Work Product Suite 15
GemStone Facets and Application Servers 15
Stand-alone GemStone Facets 16

GEMSTONE®

What Are the Technical Challenges?

Applications on the forefront will handle distributed business processes faster and smarter. But to do that, all the needed information and business logic must be available to every person and application involved in the business process at the instant transactions are happening. Traditional Java application architectures are ill equipped to deliver scalable, multi-user applications because their handling of process data presents a number of challenges that cause inherent delays in application response.

Avoid Data Conversion

Traditional application architectures rely on back-end systems—databases, file systems, and back-office systems—for management and transaction control of data used by applications. Data in these back-end systems is kept in the format of that system: relational data must be mapped, XML and flat file data must be parsed. But Java applications work with Java objects. For some applications, populating application objects may involve complex multi-dimensional JOINs, causing significant development effort and huge delays in application response. The performance hit of these complex JOINs can be prohibitive for some real-time applications. Besides delaying the application response at runtime, this on-going retrieval and conversion duplicates effort, takes up network bandwidth, and increases the load on system CPUs, further inhibiting performance.

To avoid the complexity and compute time of these JOINs, applications and application servers using CMP may choose to construct one table per object. This means that objects must be designed so that they can fit into a single relational table. The result is that the ensuing code may forfeit the benefits of an object design by requiring an object design that is not based on the problem space being solved, but is based on the way relational data is stored. Populating the application may mean retrieving a multitude of pre-constructed, object-like relational tables that result in code becoming incomprehensible. For example, instead of having a single object graph to represent a car and it components (tires, hubcaps, engine, sparkplugs, etc.), the application will have a separate object for each piece and the application code must construct the relationships amount the objects. The resulting code is both inefficient and harder for a person to understand.

Managing Large Object Graphics in the JVM

Large query results or other data sets can result in an excessive virtual machine (VM) footprint to store the data set and materialize the objects. Those sets that are too large to be stored in memory can cause repeated trips to back-end sources. Furthermore, distributing more of an object graph to the client then it needs wastes time and system resources, and it may increase maintenance costs by forcing the client to deal with schema details beyond its needs. The ideal performance could be obtained if large object graphs could be faulted into a VM as objects in the graph are referenced. This would prevent parts of the graph that are not being used from consuming the compute time necessary to materialize them from the back-end and from consuming memory in the VM.

Avoiding Redundant Reading of Persistent Objects

Problems of information latency are multiplied when processes running on different virtual and physical machines repeatedly retrieve and convert the same data through remote calls to back-end systems. The same information must be pulled from the database multiple times because each process needs a copy. There is no transparent way to share information across processes. This results in contention for transactional access to the data, additional load on the back-end source, and unnecessary network load.

Sharing Live Information across Virtual and Physical Machines

Current Java business systems typically are deployed on Web servers, J2EE application servers, and Java messaging systems. It is critical that information distributed among these servers be synchronized, both for availability and to ensure consistent business results. Typical Java architectures keep information in sync across applications, machines, and back-end data sources by serializing data and broadcasting it across a network. This results in latency that affects overall system throughput and scalability. The traffic of synchronization competes for network bandwidth with the traffic of back-end data retrieval, further straining system resources.

Reducing Time in Transactions

When applications rely on back-end data sources for transaction control, data must be converted back and forth between Java objects and the back-end format for every transaction. This places additional load on back-end resources that may be stressed already. Competition for transactional access to the same back-end data may make it necessary to wait for locks or risk commit failure. All of these factors can delay transaction processing and degrade application responsiveness.

GemStone Facets: Real-world Technology for Real-time Results

GemStone Facets is the industry's first shared transactional Java cache, software that boosts the power of traditional Java architectures to dramatically enhance the performance of complex, distributed applications. Applications deployed on Web servers, J2EE application servers, and Java messaging systems use GemStone Facets to achieve real-time performance by providing applications with local management of working data. This frees them from the delays of repeated communication with back-end systems and the overhead of synchronizing data among distributed servers.

Any Java application, regardless of whether it is a JSP, a servlet, an EJB, a JMS system, or simply a Java application can integrate seamlessly with GemStone Facets to yield immediate performance gains (see Figure 1). With J2EE application servers, GemStone Facets leverages the Java Connector Architecture (JCA) to function as an integral part of the EJB container, gaining security and transaction information directly from the container itself.

GemStone's patented caching technology provides thousands of users and multiple distributed applications with a logically consistent, recoverable, highly performant view of critical business data across islands of information. With GemStone Facets, developers can use ordinary Java objects to quickly build strategic, real-time applications that were never practicable before. No special subclassing is required.

The VMs use Facets' unique Persistent Cache Architecture (PCA) to share persistent Java objects, thus supporting transparent processing distribution. Each VM supports full transactions on objects in a small VM memory footprint, supplying the headroom needed for complex business logic. GemStone is a source licensee for the Sun JDK, and our enhanced standard Java VM has been certified Java Compatible. All Java APIs are available to every Facets VM and are preloaded into the Facets Java repository.

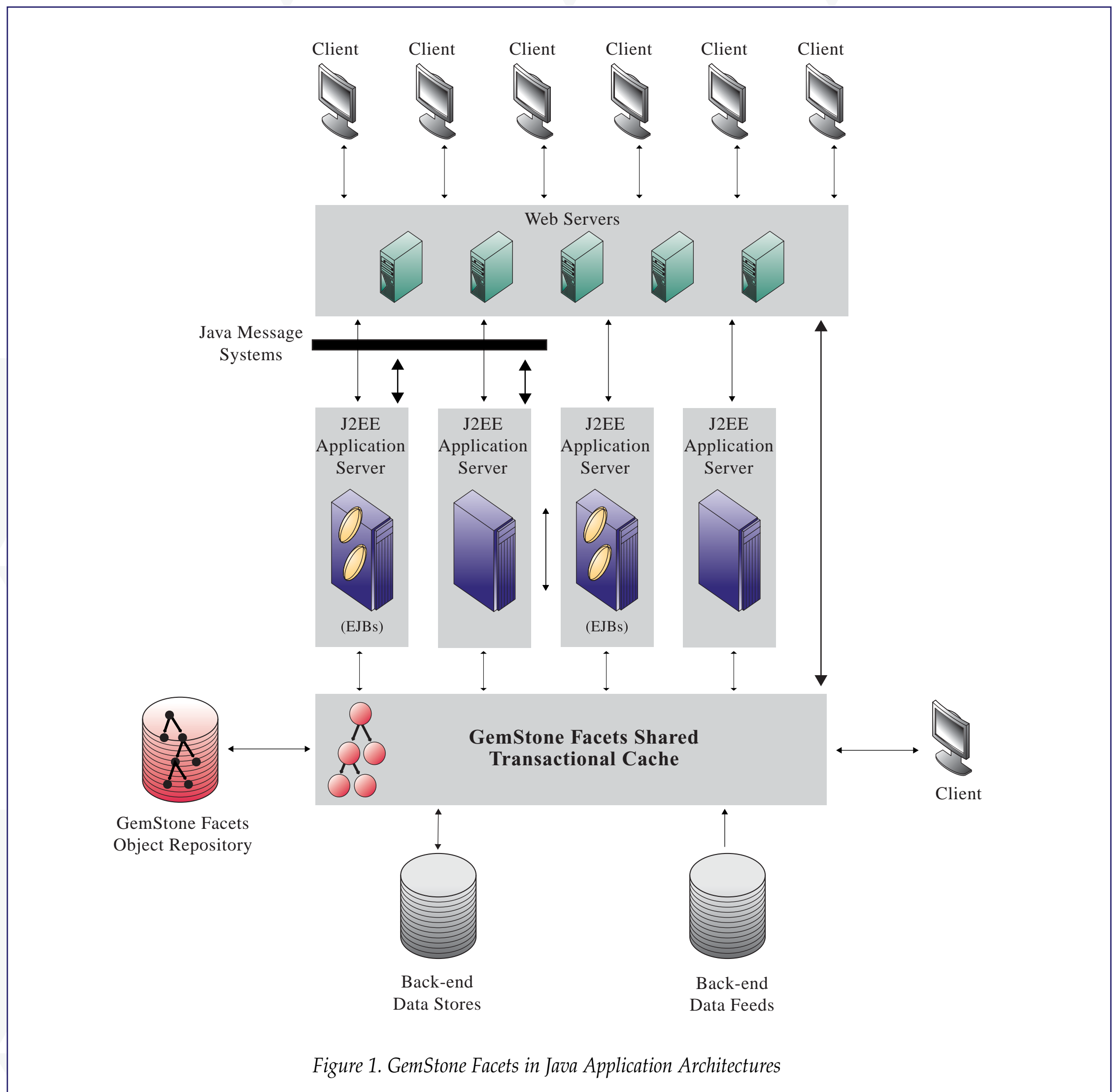

Figure 1. GemStone Facets in Java Application Architectures

Architecture Of A Real-Time Engine

GemStone Facets' architecture centers on a shared transactional Java workspace, which is a distributed, transactional, shared memory cache (see Figure 2). This shared cache boosts application performance by keeping large working sets of objects available to distributed applications while maintaining transactional consistency.

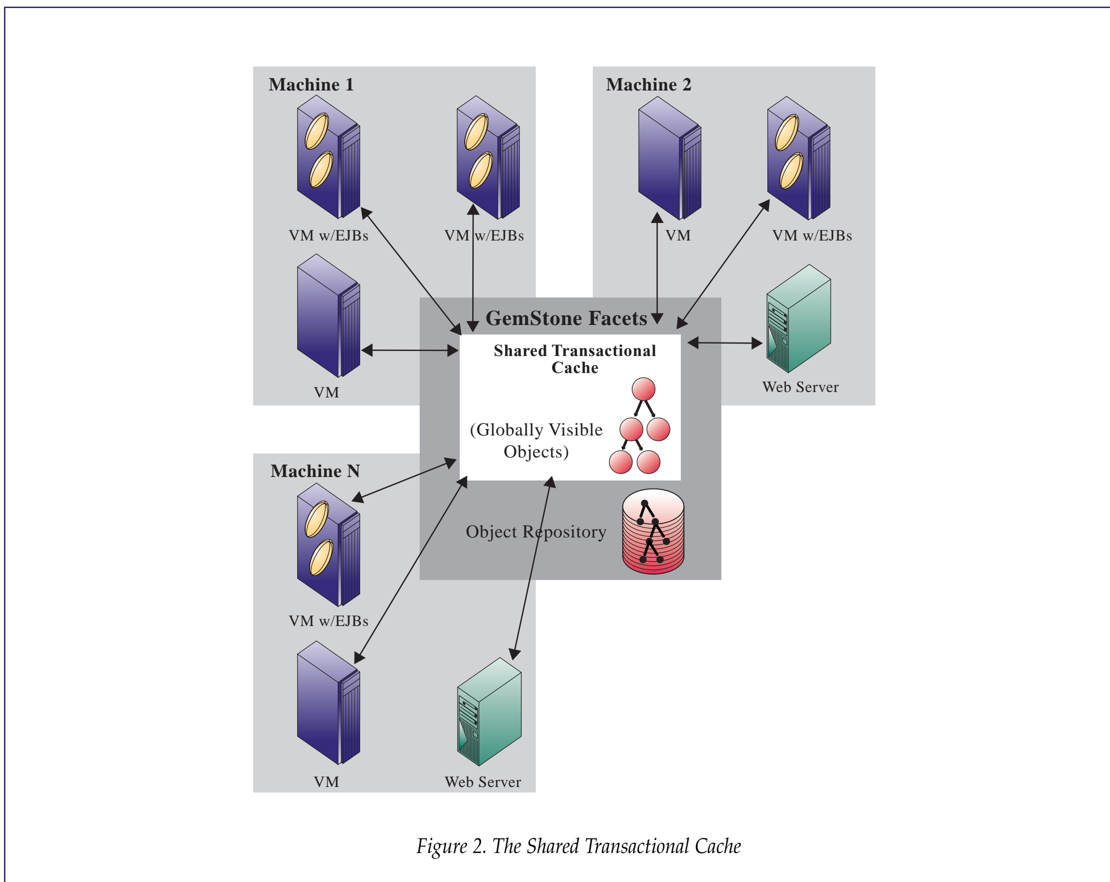

Figure 2. The Shared Transactional Cache

The GemStone Facets shared transactional cache provides three fundamental facilities to an application designer:

Shared data across multiple VMs and machines: A distributed shared cache enables applications to share objects across many concurrent threads on different VMs and different physical machines, thereby supporting thousands of concurrent user interactions and millions of transactions per day. Instead of multi-casting copies of data, user sessions and other processes access data through a shared memory cache on each machine, reducing remote communication.

Object-level transaction control: When applications rely on back-end data sources for transaction control, data must be converted to back-end formats for every transaction. Transaction control at the object level minimizes round trips to back-end data sources, allowing back-end storage to be handled independently and asynchronously while still guaranteeing the semantics of ACID transactions.

Scalable, fault-tolerant object storage: The shared transactional cache transparently interfaces with a vast, disk-based Java object repository capable of handling billions of objects and hundreds of gigabytes of working data. The repository provides both native object persistence and support for the Java Data Objects (JDO) standard persistence mechanism. Any Java object referenced by a persistent object—whether accessed through EJBs, JSPs, servlets or pure Java—is automatically made persistent, with minimal coding effort required. Using the repository, applications can assemble large data sets from distributed sources and use them as needed, avoiding the overhead of repeated access to back-end systems.

These facilities are supported by an industrial-strength infrastructure:

High availability: GemStone Facets supplies high availability from the process to the system level. GemStone Facets' out-of-the-box fault tolerance continuously monitors and automatically restarts processes. In case of total system failure, GemStone Facets allows for easy failover to a backup system. GemStone Facets supports NAS and SAN disk devices for object extent and transaction log storage. For complete high availability, GemStone Facets also supports third-party clustering software.

Security: With the JCA 1.0 API, GemStone Facets supports J2EE security. Features include basic password authentication, mapping users to roles, and access control lists (ACLs). JCA also propagates the JAAS subject established by the application server and allows for credential impersonation. GemStone Facets is also compatible with other security mechanisms, including user name/password or certificate-based authentication, pluggable PKI-based interfaces, X.500 names, X.509 certificates, SSL 3.0, and a pluggable interface to any SSL provider.

Administration and performance monitoring tools: GemStone Facets' GUI-based administration tools include a Performance Monitor and an Administration Console for system configuration and management. A full administration API is available for scripted administration tasks. GemStone Facets also includes a Visual Statistics Display (VSD) for precision tuning of application performance. Statistics are generated automatically by all system components and then stored in shared memory for maximum efficiency. When statistics collection is activated, statistics are asynchronously written to the system disk to be monitored in real time or at your convenience. The VSD enables system administrators to correlate events in the statistics stream, identify bottlenecks, and tune accordingly. GemStone Facets also includes a browser for monitoring the persistent objects in the repository.

How Does GemStone Facets Help?

When applications must support large user communities in near-real time, every microsecond counts. GemStone Facets offers a variety of approaches to meet today's challenges of reducing latency and improving application performance.

Avoiding Data Conversion

In some situations, the overhead of data conversion can be eliminated entirely. A running application often generates intermediate data that might be characterized as application metadata, or data that is used to manage work in process as it progresses through the system. This metadata has a relatively short shelf life and is not data that needs to be stored in a database of record. Metadata can be cached in GemStone Facets while it is of interest, then removed when it is no longer needed. Similarly, many applications involve data that has a limited shelf life of days or months. For these applications, it makes sense to cache the data in GemStone Facets while it is in frequent use, then write it out to a data warehouse.

Objects in the shared transactional cache are automatically persisted in the repository if they are reachable from an object that is currently stored there. (This is called *persistence by reachability*.) When an object is retrieved from the repository, it appears instantiated with its state initialized, all without any mapping from the format of the back-end system, whether that is a flat file system, relational database, or back-office information system. An application can drop an object into the shared transactional cache and pick it up later without creating any extra application code to manage the persistence.

With persistence in GemStone Facets, development and maintenance are simplified and streamlined. Mapping code is required only for data that needs to be written to a back-end system. Because applications do not depend on back-end systems for transaction control or persistence, applications can be developed quickly using the shared transactional cache for transparent persistence and distributed transaction control.

Persistence in GemStone Facets also frees applications from dependence on back-end schemas. With object-on-object storage provided by GemStone Facets, back-end schemas and mapping can be changed in the background and integrated with the application when ready.

Managing Large Object Graphs in the JVM

GemStone Facets overcomes memory limitations with an object repository capable of handling billions of objects and gigabytes of working data. Using the GemStone Facets repository, applications can assemble large data sets from distributed sources and use them as needed, avoiding the overhead of repeated access to back-end systems. Alternatively, developers can use GemStone Facets as the store of record. The shared cache keeps often-used objects in caches local to the VMs that use them, and manages the rest of the objects on disk until they are needed. Scalable collections technology lets applications efficiently handle data sets with millions of entries.

Initially, only the top-level object in the cache is materialized in the VM's workspace (see Figure 3). Network traffic is kept to a minimum because other objects are materialized only when needed. When a process attempts to reference one of these other objects, Facets recognizes that another object should be load it into the cache.

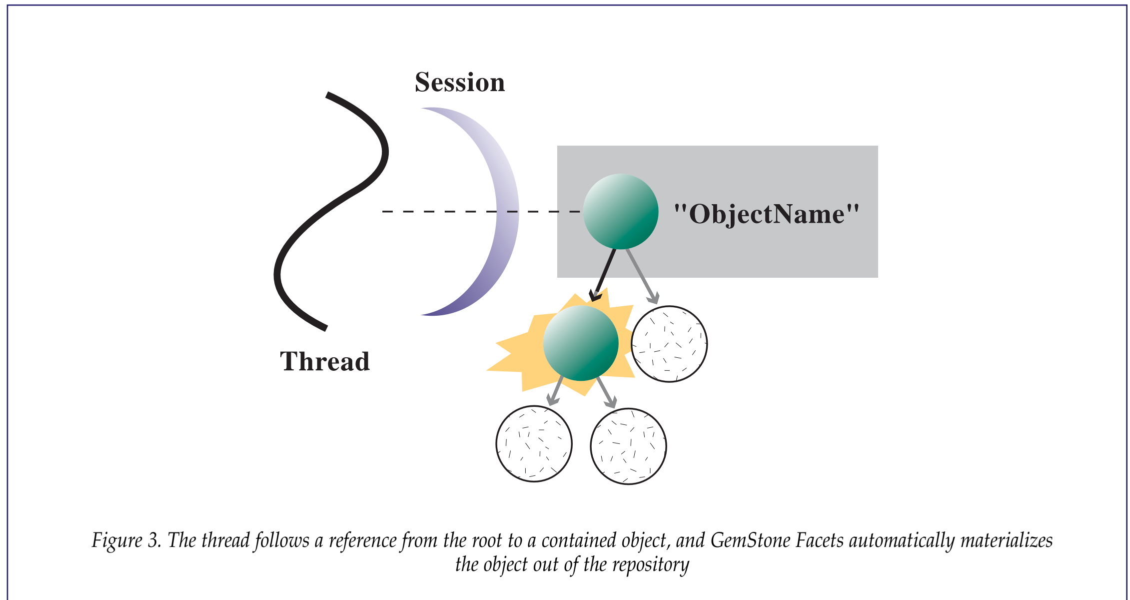

Figure 3. The thread follows a reference from the root to a contained object, and GemStone Facets automatically materializes the object out of the repository

The GemStone Facets scalable collection classes have a much more efficient "memory footprint" than their java.util counterparts for large collections (> 20,000 elements). Several of the java.util collection classes use a single large monolithic Java array—collections of hundreds of thousands of elements require the entire array to be in virtual memory at one time. The GemStone Facets counterparts to these classes use many smaller array "chunks," allowing incremental growth rather than copying very large arrays. Not all of these array chunks need be in memory at one time, so applications that only access and update a subset of elements only need to read in the portions of the array that contain those elements.

Avoiding Redundant Reading of Persistent Objects

A primary use of GemStone Facets is as a cache to improve application performance. Because GemStone Facets stores massive amounts of data in object form, much of the ongoing overhead of converting working objects to back-end data sources can be avoided. GemStone Facets offers a means to keep these sometimes massive data sets in object form, local to the processes where they are needed, and safe in case of system disruption. The application can deal with native, transactional Java objects rather than cumbersome SQL statements. Consider, for example, an insurance application involving a large ratings table, data that is used very frequently in the application but is never modified. If information is in back-end systems, it has to be read and mapped in and added to the ratings table each time a user needs some data. This data could be pre-cached when the application starts up, eliminating on-going mapping overhead and the need to page in the data bit by bit. Once the data is in the cache, wait time for data retrieval is eliminated.

Sharing Live Information across Virtual and Physical Machines

Today's Java application may be scaled by adding more server VMs, more host machines, or both. Application tasks may be partitioned across separate VMs specially configured for each task, and discrete applications running on separate VMs may be combined to support cross-functional processes.

With GemStone Facets, processes running on distributed Java VMs can all see and manipulate the same objects through connections to the shared transactional cache. Each connection provides a transactional view into the shared transactional cache's repository of possibly billions of objects. GemStone Facets transparently manages caching of objects in the application's transactional view (see Figure 4). The developer writes code as if all the needed objects were always present.

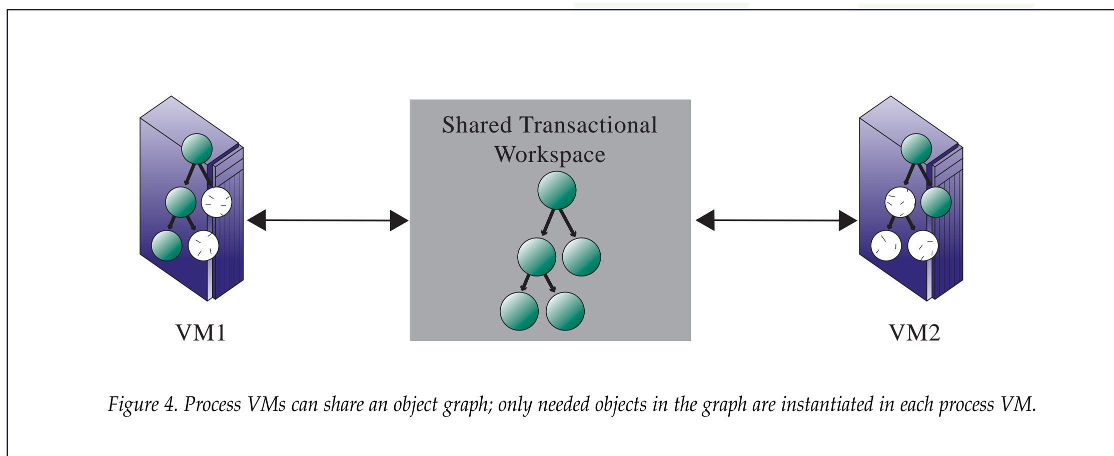

Figure 4. Process VMs can share an object graph; only needed objects in the graph are instantiated in each process VM.

By using GemStone Facets, the power of a shared, distributed, transactional cache using ACID semantics is unleashed across the application VMs. Data that is currently kept in sync by point-to-point or UDP broadcast communications can instead be entrusted to GemStone Facets, where it is efficiently distributed by the Java virtual machine's object manager to other nodes in the cluster.

When a transaction is started on a Facets connection, the system ensures that modifications made by other concurrent transactions are not seen by the application. The application sees a consistent view of all shared objects and performs its processing in isolation. When the transaction is committed, any changes made to shared objects are incorporated into the current state so that they can be seen by other applications (see Figure 5).

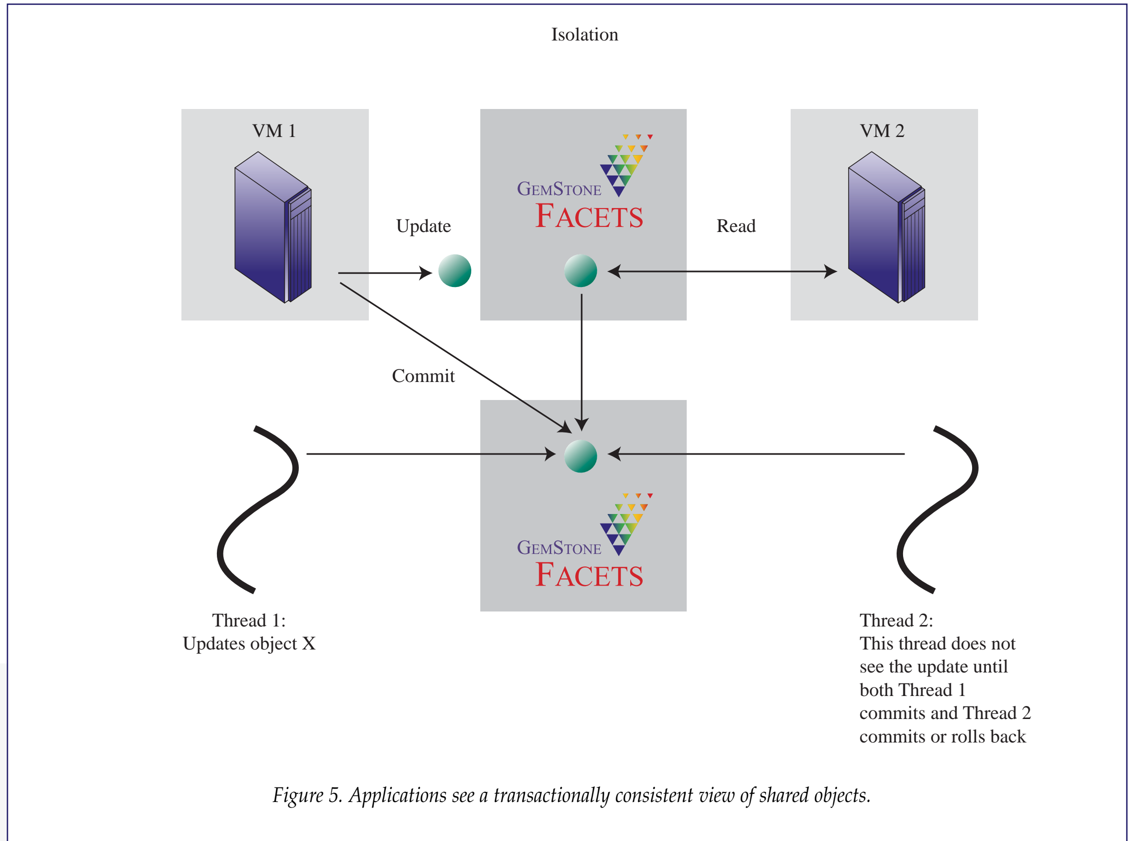

Figure 5. Applications see a transactionally consistent view of shared objects.

Reducing Time in Transactions

GemStone Facets includes specialized concurrency control mechanisms that help applications to handle thousands of concurrent users. Optimistic concurrency means no waiting on locks, a clear performance win for an application. GemStone's patented concurrent update (CU) technology allows designers to create applications with optimistic concurrency but also with the guarantee that transactions will commit successfully. For example, CuQueue allows any number of clients to add objects concurrently to a structure that appears to the application as a normal queue. All clients are guaranteed that their commits will succeed so long as only a single client removes objects from the queue.

Another performance-enhancing approach is to use GemStone Facets to separate transaction control from back-end data conversion. In many applications, updates to the back end may be delayed a fraction of a second, a few seconds, or even hours as long as data is kept transactionally consistent and safely stored on disk. An order tracking system might be a good example. Updates to orders can be made and committed in GemStone Facets, so that the application can proceed with other work. The results can be pushed or pulled to a back-end system such as an RDB a few seconds later if necessary. System responsiveness is improved for users, data integrity is preserved, and critical data is stored securely in the database of record.

Integration into J2EE Applications

When it is integrated with a J2EE Application Server, GemStone Facets provides a sharable virtual store for active Java objects that can be used directly from Session beans or to implement BMP Entity beans. Facets provides fast access to objects across the application server cluster through its shared transactional cache repository extending from local Java object memory through distributed shared memory to durable, transactional storage on disk. A J2EE Connector Architecture (JCA or J2CA) adapter seamlessly integrates Facets into application servers supporting JCA 1.0, including BEA WebLogic™, IBM WebSphere™ and JBoss™. J2EE applications can use Facets to share native Java objects with other servers across the application server cluster as well as with non-clustered servers.

GemStone Facets contains a Java Development Kit and a J2EE Connector Architecture Adapter that are used to enable access to Facets from the application server. The application server startup sequence is modified to use the Facets JDK. The Facets JCA adapter is then deployed and configured using the application server's deployment tool (for JBoss, Mbean XML is provided by JBoss for insertion into jboss.jcml).

Deploying the adapter creates a ConnectionFactory that is placed in the server's naming service where it can be accessed by applications. The ConnectionFactory is used to obtain a pooled Connection object that holds a JNDI context. This naming context is the root of shared objects in GemStone Facets. The Connection is transactional and participates as a resource in JTA transactions.

Inside the application, beans access shared, persistent information by looking up the GemStone Facets ConnectionFactory in the container's JNDI. The container automatically configures the factory for the proper Facets system and enlists Facets connections with the container's transaction manager. Facets also automatically receives caller and application security information from the container. To the application, Facets appears as, among other things, a persistent JNDI that is available across the application server cluster as well as in other applications.

The GemStone Facets JNDI can be used to pass information from one application server to another or to record information for use later by the same server. Since the JNDI implementation is in the same process as the application server, no network communications overhead is incurred. Facets transparently manages the propagation of changed objects between server VMs.

Accessing Objects in J2EE Applications

GemStone Facets organizes shared objects into a JNDI directory. The initial context of the directory is obtained from the Facets Connection object and is used like any other JNDI context. Objects bound into this JNDI become shared, transactional and durable. This transformation is performed on a bound object when the current transaction is committed and, by default, is performed on all objects reachable from the bound object. The rules are similar to serialization, though Facets does not use serialization to

manage objects. Facets hooks into the Sun JVM's object manager to perform this feat faster than is possible with serialization schemes or even Aspect Oriented Programming techniques.

Controlling Transactions in J2EE Applications

The GemStone Facets JCA adapter is fully transactional and supports two-phase commit for distributed transaction processing. Transactions are controlled as with a JDBC connection. J2EE applications can have transactions configured in their deployment descriptors. Facets also supports local transaction processing and supports the JTA's UserTransaction API.

Conclusions

As business systems grow more complex, they must allow thousands of users to act as one, working with complex information that changes moment by moment. Traditional application architectures cannot meet the immediacy of today's information needs.

GemStone Facets provides a non-intrusive means to overcome the performance barriers of traditional architectures. It provides all standard Java 1.42 interfaces, such as JNDI, JTA, JDO, and JCA. It seamlessly integrates with application servers. It simply runs in the back end, providing all the advantages of its shared transactional cache:

- Transparent persistence mechanisms for Java objects can reduce or eliminate the overhead of O/R mapping, speeding development time and system responsiveness. No restrictions are made on the structure or complexity of the user's object model, and developers don't have to subdivide their models into sections for persistence implementation. Working data and metadata need never be mapped to back-end data stores.

- The vast GemStone Facets object repository allows developers to assemble complex object graphs from diverse sources, maintaining object relationships without requiring everything to fit in memory and without using serialization to store objects in data files. Developers write code as if all the needed objects were always present in the application's transactional view. GemStone Facets transparently manages the persistent objects, efficiently materializing them in the VM's memory only as they are needed. Clustering of objects on the disk can aid in prefetching objects for even higher performance.

- Inherent caching capabilities allow persistent objects that were constructed from diverse sources to be accessed repeatedly without recurring retrieval and conversion costs. The back-end systems can be updated as needed, and some architectural applications of GemStone Facets allow for asynchronous write-through caching.

- GemStone Facets' multi-VM architecture works seamlessly across multiple virtual and physical machines. Processes running on distributed Java VMs can all see and manipulate the same objects through connections to the shared transactional cache. Each connection provides a transactional view into the shared transactional cache's repository of possibly billions of objects. GemStone Facets keeps the shared transactional cache synchronized across physical nodes. Object interfaces remain the same from machine to machine. Locking and transactions also occur seamlessly on multiple machines.

- GemStone Facets supports both optimistic and pessimistic concurrency controls. Its patented technology provides optimistic concurrency control, which allows critical business data to be updated without locking. This improves performance since applications can make full use of these resources. Thousands of concurrent users can share data in real time. Applications are also easier to design with GemStone Facets because the underlying system handles the concurrent updates. GemStone's concurrent update (CU) technology enables designers to create applications with optimistic concurrency but also with the guarantee that transactions will commit successfully. CU classes allow multiple users to update the same collection object in different transactions without conflicts.

The GemStone Facets shared transactional cache provides dramatically improves performance, simplified application design, and fault tolerance of critical business data. It complements the functionality of Web servers, J2EE application servers, and Java messaging systems by transparently distributing and managing needed objects in its distributed shared cache. This simplifies development of distributed multi-VM architectures because developers know that no matter where a process runs, the same global object cache is available.

GemStone Facets has been decades in the making. It is based on proven technology that customers are using today to create real-time performance for sophisticated, high-return business applications. Organizations using GemStone Facets can seamlessly share real-time data across users, applications, and systems, creating an environment in which business processes run faster, business decisions are more accurate, and work can be accomplished at a lower cost. For more information on GemStone Facets, please visit http://www.gemstone.com

Application Usage Scenerios

Facets can be used to advantage in a variety of applications, such as the system architecture shown previously in Figure 1. Facets can be seamlessly integrated to provide the essential shared transactional object cache technology as part of a state of the art software architecture.

GemStone Facets as a Cache

A primary use of Facets is as a cache to expedite applications. Because Facets stores objects in object form, the usual overhead of object-relational (O/R) mapping can be avoided. For example, large amounts of data that is used in a read-only fashion can be pre-cached at application startup. This mitigates wait times for users while needed data is fetched. For other applications, laboriously mapped data may have a limited shelf life, perhaps days or months, so it makes sense to cache the data in Facets while it is of interest and later either remove it or write it out to a data warehouse.

Still another style of caching is used by applications with an RDB as the database of record that wish to cache data in GemStone Facets for performance reasons, and then allow updates to be made in Facets that are afterward written through to the RDB. These applications can often allow the write through to occur asynchronously from the commit in Facets. An update delay of a fraction of a second, a few seconds, or even hours may be acceptable depending on the semantics of the application. An order tracking system is an example of this kind of system. Updates to the order are made in Facets, then pushed into the RDB a few seconds later.

GemStone Facets as a Work Product Store

The data that organizations run their business on is often kept in large RDBs that serve as the database of record. However, when an application is running there is often intermediate data generated that might be characterized as application metadata—the data that is used to manage work product as it progresses through the system. This data has a relatively short shelf life and is not fundamental data needing to be stored in the RDB. Facets provides an effective persistent store for such data. The data can be stored without the overhead of O/R mapping but still is guaranteed the semantics of ACID transactions. Facets therefore provides a very effective mechanism for providing fault tolerance to a system for relativity small development costs. Work in progress is simply persisted into Facets. In the event of a system crash, it can be recovered and continued.

GemStone Facets and Application Servers

Facets can be integrated seamlessly into systems being developed with application servers. By using the J2EE Connector Architecture (JCA) Resource Adapter provided with GemStone Facets, the application is able to look up the Facets Connection Factory in the container's JNDI in a standard way. While application servers provide needed J2EE functionality, Facets provides the essential object sharing. This frees designers to consider multi-VM architectures because they know no matter where a thread runs, the same global object cache is available. Facets is non-intrusive to application servers. It runs in the backend providing the advantages of a shared transactional object cache.

Stand-alone GemStone Facets

Yet another use for Facets is the stand-alone Facets application. In this case, Facets becomes the persistent store of record for the application—though RDBs may or may not still be present in the application. A stand-alone Facets application has the advantage that everything is an object. There is no need for an O/R mapping layer or the maintenance burden of synching this layer with changes in RDB schema. Any system that can be built with objects is a candidate for a stand-alone Facets implementation.

GemStone Systems, Inc.

1260 NW Waterhouse Ave., Suite 200 Beaverton, OR 97006 | **Phone:** 503.533.3000 | **Fax:** 503.629.8556 | info@gemstone.com | www.gemstone.com

Copyright © 2004 by GemStone Systems, Inc. All rights reserved. GemStone®, GemStone Facets™, and the GemStone logo are trademarks or registered trademarks of GemStone Systems, Inc. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All other trade names or trademarks are the property of their respective owners. Information in this document is subject to change without notice.

4/04