Extreme Clustering: The Scalable Architecture for B2B Applications

June 1, 2000

Brought to you by:

GemStone Systems, Inc.

a Brokat Company

Abstract

Scalability is the ultimate challenge of e-commerce technology. A B2B e-commerce system must provide thousands of concurrent users with real-time shared transactional access to complex business rules and data from widely distributed business systems. The Java™ 2 platform, Enterprise Edition (J2EE™) community has embraced clustering as a solution to the scalability issue. However, early J2EE clustering architectures achieve only limited scalability at the cost of excessive hardware requirements, complex system administration and excessive network traffic. GemStone/J, with its Extreme Clustering™ architecture, is the first of a new generation of application servers, providing distributed multi-level resource clustering specifically designed to scale for the high-volume, distributed B2B e-commerce environment.

Introduction

Scalability of e-commerce systems will make or break an e-business strategy. If you have it, you can scale your business through increased sales and new products and services, often with the same or fewer resources. If you don't have it, you can experience massive loss of confidence with customers and business partners in minutes or hours — confidence that can take months or years to regain. As recent "denial of service" attacks on Yahoo, eBay, Amazon.com and other Web businesses have shown, even the most established e-commerce sites are vulnerable to surges in user and transaction load.

Denial of service attacks on business-to-consumer (B2C) sites are a wake-up call. What happens when we are using the Web to do more than just buy books? The real danger for e-commerce scalability is in business-to-business (B2B) Web systems, where business relationships and profits rest on large, complex, real-time transactions. And for large-scale B2B sites, ongoing poor response or poor reliability can be far more damaging and insidious than hacker attacks. Inadequate day-to-day performance has far more potential to alienate customers and business partners, pushing them to the sites of competitors who provide faster, more reliable on-line services and support.

The B2B Scalability Challenge

For B2C Web sites, transactions are generally simple and the greatest challenge is often in getting personalized content to a browser. In a B2B system, there may not be a human on the client side to guess what to do next, or to ask if something is unclear. A B2B transaction must be much more precise, while at the same time, adapting to many different business environments. Look at the requirements and characteristics of a digital exchange, a typical B2B e-business system.

- The system must tie together information such as product catalogs from many suppliers and applications such as order management, scheduling, and auctions in real time.
- The site must implement complex business rules to customize business transactions between customers and suppliers.
- It must scale to service thousands of concurrent customers initiating complex transactions with information drawn from hundreds of partners and suppliers.

- The development team can't rely on pre-packaged software to provide the complex business rules, sophisticated transaction control, or flexibility required by the application. For maximum flexibility and adherence to emerging standards, they will implement most of the business logic using J2EE technologies.
- Competitive advantage comes from non-stop availability. The application cannot be taken down for updates, performance tuning, system reconfiguration or system failure.

Complex B2B systems like this digital exchange require non-stop, high-speed, high-volume service. For these sites, scalability requirements go beyond the limited definitions of the past. In assessing scalability of a system or an e-commerce platform, you must consider:

- **Performance:** ability to handle peak loads, in terms of users, information and transactions
- **Availability:** ability to maintain service and performance all the time
- **Adaptability:** ability to maintain service and performance as the system expands to meet new functional or performance requirements

Many businesses have chosen to base their Web commerce systems on J2EE and its Enterprise JavaBeans™ (EJB) components, because J2EE is intended to be a complete platform for Web-enabled, multi-tier, secure, transactional Java applications. J2EE promises better quality, maintainability, and portability of systems and increased productivity and economic return for businesses. With the popularity of J2EE, a number of vendors are offering application servers, e-commerce platforms aimed at providing object execution engines, transactional control, security, and other system infrastructure. However, these offerings must be viewed much like relational technology a decade ago. Despite J2EE standardization, there are vast differences in the capabilities and scalability of these systems.

From Clustering to Extreme Clustering

In its quest for scalability, the J2EE world has embraced "clustering," the use of shared redundant resources as a solution. In the early days of networked computing and "business hardening," the IT world began to use redundant hardware resources to provide failover for critical business systems. Disk striping, RAID technology, and hosts such as Tandem computers with their redundant servers were developed to provide "hot backup" in case of resource failure.

Like the redundancy strategies of the 1980s, early J2EE clustering architectures have focused mainly on failover, with limited gains in scalability. These carbon-copy or "weak" clustering architectures offer a single Java virtual machine (VM) per host machine and keep clusters of these very limited application servers in sync at the cost of duplicate hardware and increased system administration. In these systems, clustering is less a scalability enhancement than an attempt to overcome the single VM-per-machine limitation. As load increases, these fragile systems must be taken off line to add more hosts, gaining marginal scalability at the cost of availability. While these systems may support the first tentative steps of an e-business strategy, they will never scale to support the complex distributed business processes that are the ultimate goal of B2B e-commerce.

GemStone/J's Extreme Clustering product is the first of a new generation of application servers. It offers distributed multi-level resource clustering designed for the high-volume, distributed B2B e-commerce environment, able to support tens of thousands of concurrent users and many millions of transactions per day.

Extreme Clustering™ includes:

- **Multi-VM Architecture:** A distributed clustering architecture that transparently manages hundreds of VMs and other resources per server and across multiple servers without creating additional burdens on the system or the business
- **Smart Load Balancing™:** Intelligent, multi-level load balancing that matches processing needs for specific operations to Java VMs configured to meet those needs
- **Total Availability™:** Multi-level failover to ensure continuous service and performance, plus tools and features that allow administrators to expand and adapt the e-commerce system without interrupting service

Extreme Clustering is supported and enhanced by GemStone's Persistent Cache Architecture™ (PCA) which enables users and processes to efficiently share access to business objects, and its Object Transaction Service which provides flexible, multi-level transaction mechanisms to efficiently maintain transactional integrity across distributed business systems (see Figure 1).

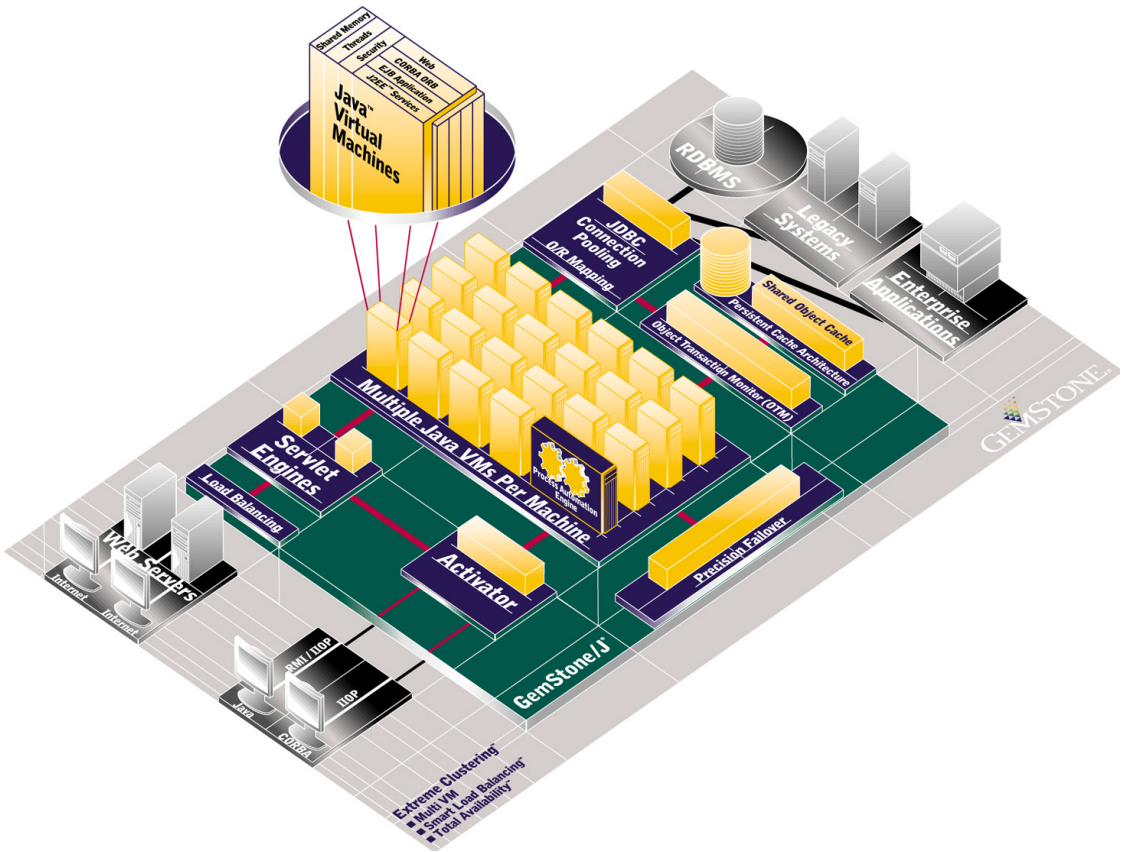

Figure 1: GemStone/J Extreme Clustering.

Extreme Clustering overcomes the B2B scalability barrier:

- Through pooled resources and intelligent load balancing, a GemStone/J system on a single high-end host can process millions of Web hits per hour, serving thousands of concurrent user requests. Because its multiple VM architecture scales further on a single platform, GemStone/J is an order of magnitude more efficient in using hardware resources, costing 2-10X less than competing application servers in a deployed high-volume Internet commerce site.
- Multi-VM architecture coupled with GemStone's transactional object cache allows GemStone/J to maintain data integrity efficiently for shared objects across VMs, multiple servers and across multiple, heterogeneous databases — a must for distributed B2B environments.
- Whereas other systems fail over only at the hardware server level, GemStone fails over at the level necessary to maintain service (VM, server, host), eliminating the need for expensive, carbon-copy redundancy. And GemStone supports maintenance, system upgrades, and dynamic reconfiguration without taking the system offline.

Any application server that does not yet include these capabilities will put e-business projects 18 to 24 months behind the technology curve and push the companies that use them beyond their windows of opportunity. It is non-trivial to integrate these technologies in a scalable way.

B2B Scalability through Extreme Clustering

Let's look at B2B scalability challenges from a system perspective. Consider the tasks and system resources involved in a single Web hit:

- B2B e-commerce sites generate their content much more dynamically, using information assembled from a variety of applications and real-time sources. For example, a digital exchange page might present product descriptions and images from one or more supplier databases, custom pricing calculated for each customer according to complex business rules, and availability and shipping information from back-end business systems at supplier sites. In J2EE, where human interfaces are involved, the dynamic content is generated by Java servlets or Java Server Pages (JSPs), so a servlet or JSP must be initiated when a client hit arrives at the Web server (see Figure 2 on next page).
- To generate its dynamic content, the servlet must either access data directly out of a database or delegate these responsibilities to an application component (typically, a set of Java beans). Invoking a component incurs the overhead of component creation.
- Pulling data out of the database involves querying, transporting the result set into the object world across a JDBC interface and doing the O/R (object-to-relational) mapping necessary to populate the needed objects.
- Depending on the application, it might be necessary to retrieve data from one or more legacy systems — each represented by a separate interface requiring more component creation.
- The hit might result in the need to commit data into the database. If several backend data sources were involved, this would incur the extra overhead of a distributed, two-phase commit.

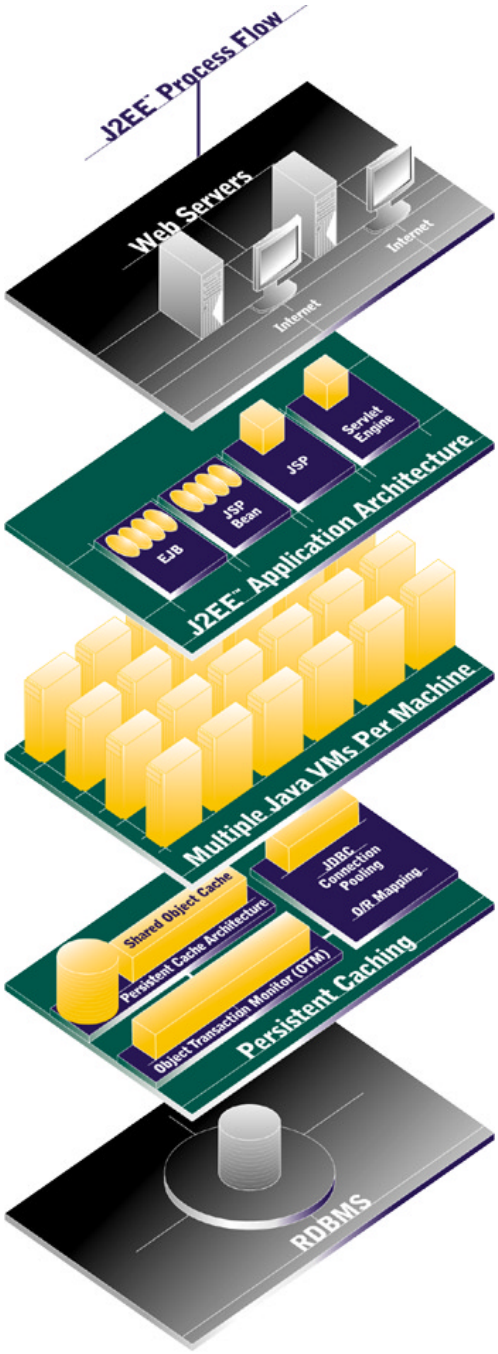

Figure 2. Servicing a Web Hit in J2EE.

- Finally, methods in the objects are exercised to produce the HTML stream in the servlet that must then be routed back through the Web server and across the network to the client.

Consider the points of resource contention (potential scalability problems) when you have many millions of Web hits every hour.

- Communication between clients, Web server(s), application server(s), and back-end databases creates network traffic.
- Web servers must assign incoming client requests to Java servlet engines for processing.
- Servlet engines rely on components executing in the application server to retrieve data from back-end databases, execute business logic on business objects, and return content for HTML pages.
- Depending on transaction models and mechanisms, shared objects and relational data can be points of contention.

GemStone/J's Extreme Clustering is designed with clustered resources at multiple levels to avoid resource bottlenecks.

- At the front end, requests from clients' Web servers are handled via routers and DNS round-robin scheduling.
- GemStone/J uses HotSpot VMs, which provide a 3-5X performance improvement over classic Java VMs through improved thread handling and garbage collection.
- Each application server host supports multiple servlet engines, each in its own Java VM. Servlet adapters in the Web servers independently balance requests across these servlet engines.
- Each GemStone/J application server manages pools of EJB/CORBA VMs across one or more hosts. The Global Name Service and Object Transaction Monitor allow objects to be distributed and shared across multiple VMs and multiple hosts, while the Activator, using Smart Load Balancing, matches processing needs for specific operations to Java VMs configured to meet those needs.
- Pooled JDBC sessions optimize access to back-end databases.

- The GemStone/J Persistent Cache Architecture underlies and supports other resources. It provides fast, transactional access to objects through its distributed shared page cache, transparent persistence for Java objects to minimize object-to-relational translation for in-process data, and support for distributed, heterogeneous transaction control. It does not rely on Java serialization or mapping to external databases.

Cost-effective Scalability for Complex Integration

Early e-business initiatives have tended to harvest "low-hanging fruit" simple departmental applications or self-service applications such as FAQs. For these simple read-mostly applications, carbon-copy clustering provided adequate scalability. But as businesses move to automate and integrate complex business relationships and processes, early clustering architectures are proving inadequate because their single VM architectures make poor use of CPU resources. To scale beyond, at best, a couple of hundred users, they require additional machines and programming, and additional ongoing administration and maintenance. Thus they are expensive to scale, both in terms of initial deployment cost and ongoing costs of ownership.

In contrast, a multi-VM system such as GemStone/J can scale to support thousands of concurrent users on a single host machine. More important in the long run, it can support a variety of differently configured processes on a single machine, allowing diverse users to share efficient access to business objects and logic.

Scalability and Single-VM Clustering

To understand this difference, let's look at a single Java VM and how it works. Virtual machines are where Java gets its work done. They provide a platform-independent execution environment for Java code. VMs create a multiprocessing environment through "threads", each of which can handle processing tasks independently.

Theoretically, a single VM can manage thousands of threads. However, in practice, a single VM rarely scales to more than a few hundred because of garbage collection, the process of reclaiming memory held by objects that are no longer in use. The more memory a VM's applications are using, the longer it takes to complete a single garbage collection cycle. A Java VM starts and stops its threads as it does garbage collection; as the number of threads increases, so does the time taken to stop and start the threads. Performance degradation begins to appear as ever-lengthening "dead zones" where the VM is unresponsive. Improvements in garbage collection introduced in HotSpot reduce the frequency of this problem, but do not eliminate it. So performance degradation due to garbage collection is a function of the amount of memory used by a VM and the number of threads the VM is running. (Some vendors of single-VM systems will claim to support thousands of users per VM. On examination, it turns out that these threads are not actually accessing database nor are they doing business processing that result in memory garbage. In other words, they are not doing any real work with real business objects.)

Experience with commercial B2B systems has shown that once a VM takes up as little as 250M of memory, the overhead of garbage collection can cause erratic response in applications, as the VM is unresponsive for a number of seconds while it performs garbage collection. Web applications might assume the system has gone down as this interruption

grows worse. In a complex B2B application, where applications are dealing with large results from SQL queries, a thread can easily use anywhere from a few Mbytes to tens of Mbytes of memory, so a VM can be overwhelmed with far less than 100 threads. In other words, a single VM could support fewer than 100 users. In many cases, the VM hits its limit, while the CPU is underutilized.

Single-VM architectures try to address this limitation through simple clustering: adding redundant servers on additional host machines. Unfortunately, while these additional servers and machines do offer system-level failover, they offer limited scalability at a high cost in hardware, software, system management, and network traffic.

With single-VM clustering, each VM (host machine) is backed up by one or more other machines running identical software, configured identically. The state of VMs/machines in a cluster is kept synchronized by serializing[1] Java objects from the primary server to the secondary server(s). A significant number of CPU cycles on the primary machines is spent serializing objects. A significant portion of the network bandwidth is spent transmitting, and CPU cycles on the secondary machine have to be used de-serializing the objects. (Some weak clustering architectures do allow more than one VM per machine. However, each VM requires a separate IP address and functions as remote from other VMs on the same machine. This approach still requires serialization, and it is a headache for system administrators.)

Now say, for example, you had a primary server whose VM is already large and can't support more users. (The CPU could support many more users, but the single VM cannot.) You could buy a second machine (with a second software license, another IP address to manage, etc.) then set the second machine up in a "carbon-copy" cluster with the first. Now you would have two machines with two full VMs and no more ability to scale than you had with the first machine. To add more processing ability, plus a failover for it, you would have to buy two more machines. Then you could have four underutilized machines to handle a processing load that could easily be handled by one machine using an application server with a multi-VM architecture.

Cost-effective, Flexible Scalability with Multi-VM Clustering

To support thousands of concurrent users in real time with the complex application mix required by B2B systems, e-commerce platforms need a multi-VM architecture. On a single high-end host, a multi-VM server such as GemStone/J can support thousands of concurrent user interactions and millions of Web hits per hour.

Rather than running redundant VMs which must be kept in sync at all times, a multi-VM server maintains pools of VMs, each tuned to the processing needs of different use cases or applications. For example, if a particular application, such as a product catalog, required a VM to work with the results of large SQL queries, that application could be assigned to a set of VMs configured to run with no more than 10 threads. Each VM could then grow to the

[1] To be sent to remote systems, Java objects must be serialized; that is, all the data and methods must converted to network-transportable chunks and converted back to objects on the receiving end. Default Java serialization can be inefficient for complex objects. Serializing and deserializing correctly requires object-specific programming and is difficult to do for complex object networks.

optimal number of threads to make use of its CPU resources without degrading performance due to garbage collection. VMs have many configuration options, all of which add to their performance: optimum number of clients, Java heap size, lifespan before they are recycled, JDBC connection pools they will use, profiling configuration for benchmarking and tuning. And without the single-VM per machine limitation, a host machine can run more than one version of a multi-VM application server, so applications can be developed and deployed on the same machine, if desired.

With a single-VM architecture, processes with different requirements must be routed to different machines, generating extra network traffic to keep per-machine VMs in sync with each other and with data in back-end databases.

Load-based versus Connection-Based Load Balancing

To make optimal use of clustered resources, you must be able to accurately balance the processing load across those resources. First-generation application servers balance requests across clustered resources using a simple round-robin strategy or, at best, connection-based load balancing (i.e., how close is the VM to its maximum number of connections). However, load balancing is a complex issue, because some processes are much more computationally intensive than others are. Consider our digital exchange scenario: one process might simply retrieve information on a particular item for sale, while another might compute custom pricing for many items based on an agreement between the customer and supplier, the customer's purchase volumes for the month, and the total dollar value of the current order. With diverse computing demands, the best strategy is highly dependent on the application.

Next-generation clustering strategies use multi-level resource pools and intelligent, load-based balancing to distribute requests across resources. Figure 3 shows the GemStone/J load balancing scheme.

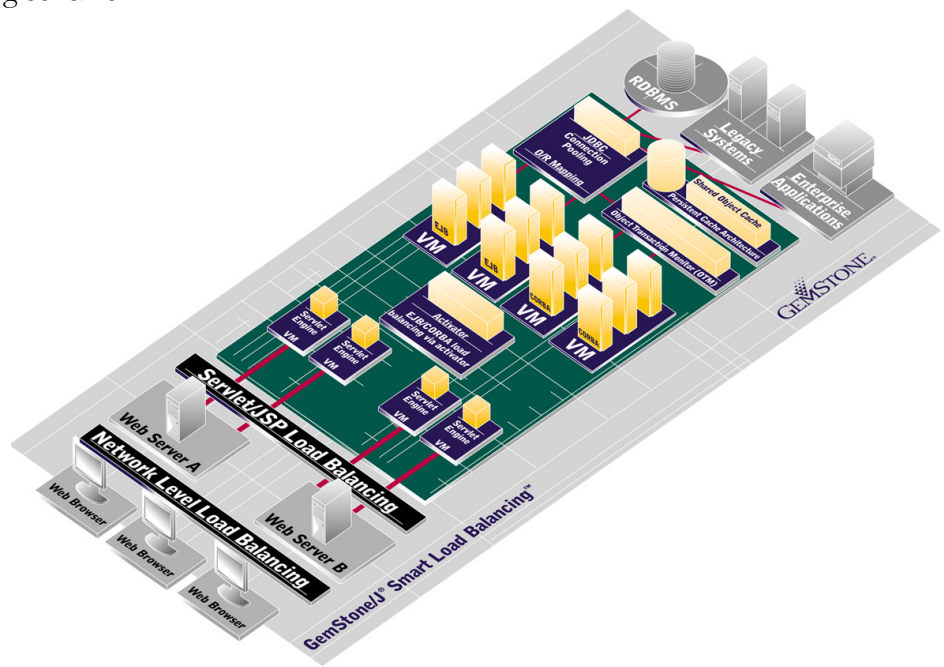

Figure 3. GemStone/J Smart Load Balancing.

GemStone/J uses pooled resources at many levels (see Figure 4).

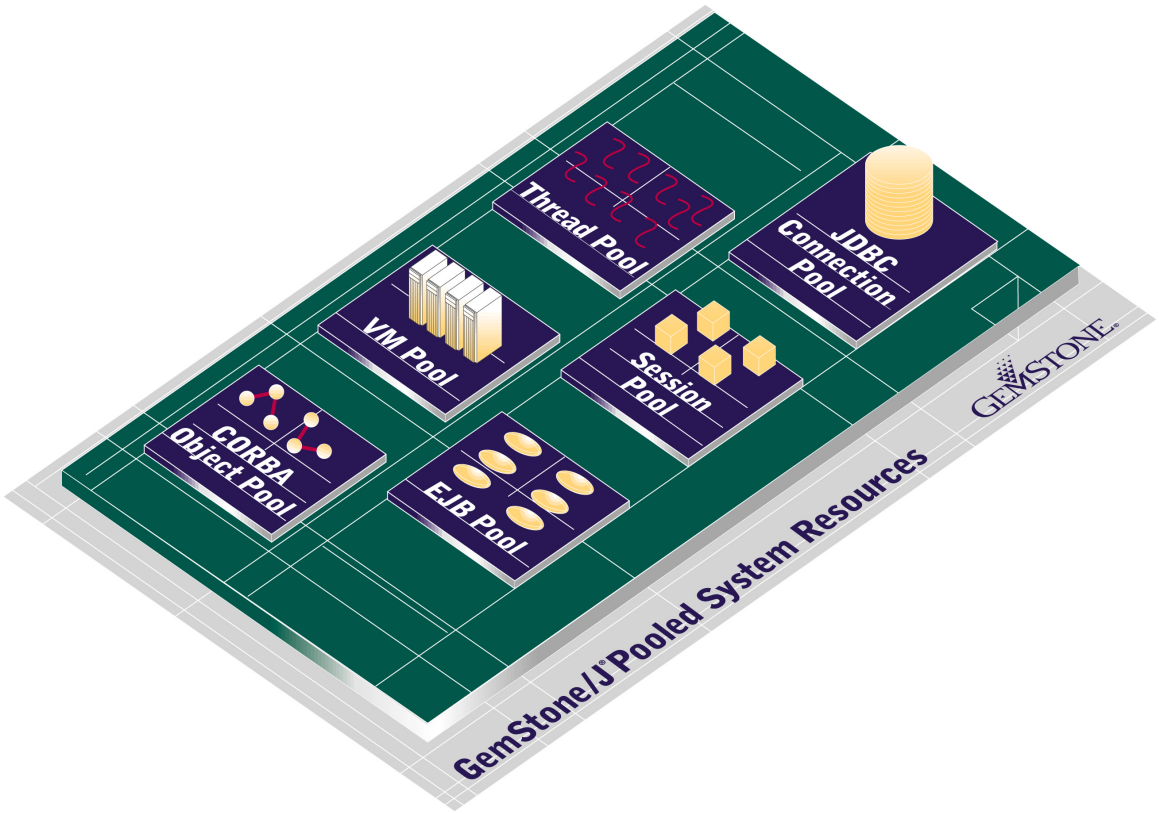

Figure 4. GemStone/J's pooled resources form the foundation for Smart Load Balancing.

- At the Web server level, network-level load balancing is handled via routers and DNS round-robin scheduling.
- Adapters in each Web server balance assignment of JSPs and servlets across pooled servlet engines running each in its own Java VM.
- Each GemStone/J application server manages pools of EJB/CORBA VMs across one or more hosts to handle the processing needs of business applications and services.
- Pooled JDBC sessions optimize access to back-end databases.
- Session state can be stored in the persistent object cache, so that clients can be routed to different Web servers, either for load balancing or in case of failure, and servers can retrieve clients' session state from the persistent cache.

At most of these levels, the amount of processing per request is small, so simple round-robin scheduling is adequate to balance the load across pooled resources. However, at the EJB level, the amount of processing can vary greatly between tasks, so the number of requests assigned to a VM may have little bearing on its actual processing load or its ability to respond quickly to another request. In order to assign tasks to the VMs that are truly able to handle them most efficiently, the GemStone/J Activator constantly monitors the workload

and responsiveness of each VM. When a new request comes is, the Activator takes into account the configuration of the VM, its workload and current rate of response in assigning new requests. If a VM's performance begins to degrade, the Activator can assign incoming requests to other VMs, starting new ones if necessary, and it can be set to replace VMs on a regular basis to avoid performance degradation.

Software Costs of Clustering Models:

Part of the point of object technology is code reuse, but some strategies, such as "weak clustering" defeat this purpose by requiring extra code. Having to create and maintain excess software delays time to market for e-business strategies, raises operating costs, and damages the bottom line.

Weak clustering servers require a great deal of excess software:

- For efficient communication, developers must write and maintain Java serialization code to synchronize redundant servers in a cluster. (Complex object networks can be difficult to properly serialize, and default Java serialization may be inefficient.)
- Load balancing is client-based, so developers must build knowledge about VM configuration into client software and change it as the system configuration changes. These load-balancing strategies are not only expensive, they are not truly dynamic.
- For each redundant server in a cluster, IT must buy and manage extra server software licenses.

A multi-VM server avoids the need for extra software, because load balancing and communication happen among multiple VMs within a single logical server. Because VMs share access to the object cache in shared memory, cross-cluster synchronization is not required, so developers don't have to write and maintain serialization code.[2] And because a single server can scale to support thousands of users, extra software licenses are not needed.

Caching and Co-location in Multi-VM Architecture

With multiple VMs on a host, application performance can be optimized by collocating components and processes to share objects and minimize communication overhead. In a multi-host, multi-VM application server, rather than each VM needing a copy of the same object, processes can share access to a single object in memory through a shared object cache on each machine. Processes in a GemStone/J application server share objects in a host's object cache.

Each VM maintains its transient local objects in local memory. However, globally visible objects in use by any VM on a host machine are located in a shared object cache on that machine (see Figure 5 on next page). Any other VM on the same machine can access objects indirectly from that shared cache, without having to retrieve the same information from an outside database. This cuts down on network traffic, on memory usage, and on CPU cycles

[2] We know of one exception to this, a vendor who supports multiple VMs on a machine and has a shared record cache. Instead of serializing Java objects, this architecture synchronizes multiple record caches using two-phase commit, an expensive operation and one that is unnecessary in an architecture such as GemStone/J.

because, again, the CPU does not have the overhead of synchronizing duplicate copies of objects. Through VM configuration options managed by the Activator, processes likely to share the same objects can be collocated on the same host to further leverage the benefits of shared objects.

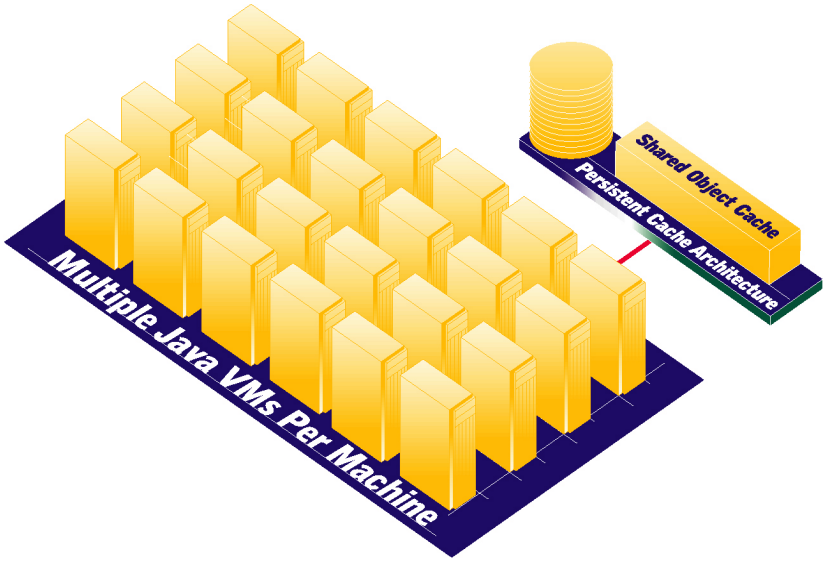

Figure 5. Multiple VMs in GemStone/J share object data stored in PCA.

The GemStone/J Activator offers a further level of collocation for J2EE performance. When one JavaBean component calls another, the second bean can be automatically activated in the same VM as the first so that all communication between the components is within the same process — the most efficient form of inter-component communication.

Multi-level Transaction Models

We have seen how a multi-VM architecture, with its pooled resources and shared object cache can enhance application performance. Of course, there is no advantage to having shared objects if there is no way to maintain transactional control at the object level, or if transaction mechanisms degrade performance. To ensure data integrity and performance, next-generation clustering architectures incorporate object transaction monitors (OTMs) as part of the application server. These integrated OTMs take advantage of shared object caches and offer flexible concurrency control mechanisms to further enhance system performance.

Object-Based Transaction Control

Simple clustering architectures offer transaction control only at the RDBMS level in a system. This is fine for data of record. However, applications typically use data in object, not relational form. Therefore, the data of record must be converted from relational to object format for use by applications and back to relational format when it is stored back into the database of record. This conversion takes up CPU cycles, memory, and network bandwidth. Here are the steps to query a persistent object out of a back-end database:

1. An SQL statement is prepared in the application to pull the correct information out of whatever tables, columns, and rows are needed.

2. The server obtains a JDBC connection to the database (or recovers one from where it was cached).
3. The application creates a JDBC statement on the connection.
4. The database executes the SQL in the context of the statement.
5. As a result of this execution, a ResultSet is returned across the interface.
6. The server must pull each individual value out of the ResultSet and assign it to the appropriate object field. This process may occur multiple times to completely populate a complex object graph.
7. After populating the state information into the objects in a graph, the server must assemble them into the correct set of relationships, so that the graph is ready for use by the application.

Application servers with state-of-the-art clustering architecture can eliminate much of this conversion overhead, and provide a level of transaction protection not found with pure relational transaction control. GemStone/J's Object Transaction Monitor manages transactions across business objects. GemStone/J provides and Object Transaction Service (OTS) that complies with the CORBA 2.3 services specification. Shared objects assembled from RDBMS data or other sources and objects created by current processes are transparently written to the persistent object cache. The Object Transaction Monitor maintains transactional views of all these shared objects: in other words, it keeps track of what process has which object, and ensures that conflicting changes are not recorded. Changes made to cached objects are synchronized with changes to relational databases in a two-phase commit.

The OTM transparently handles all objects participating in a transaction as OTS recoverable objects. Transaction logs guarantee that transactions can be recovered or rolled back in case of system failure, even if the changes haven't yet been committed to backend databases, and recovery happens automatically if the system is restarted. GemStone/J transparently wraps JDBC resources so that they are automatically under the control of the OTM. Finally, because the OTS is a CORBA service, it can also coordinate transactions with outside systems, so that the application server can provide transaction control across distributed heterogeneous systems.

Consider how object-based transaction control can free up system resources. In the course of executing business tasks and processes, applications use objects that must be kept consistent with other objects, even though the state of these objects may never become data of record. In a simple clustering system with no object transaction control, this "in-process" data would have to be mapped back to a back-end database simply to gain transaction control. In contrast, an e-commerce system with object-based transaction control can minimize conversion overhead and trips to the back-end database by exercising transaction control at the object level.

Optimistic and Pessimistic Concurrency Control

Modern computer systems achieve performance by doing tasks in parallel: multiple processor systems run multiple servers, high performance J2EE servers run multiple VMs, which run multiple threads, etc. If we could do everything in parallel, and afford enough computing power, theoretically we could perform any process instantly. In fact, there are critical tasks and information that can't be concurrent. That's why we have transaction or concurrency control. In RDBMS-based transaction control, when certain tasks are

performed, we "lock" critical objects, making them available to one process and unavailable to other processes. This locking is often fairly "coarse grained," that is, we may have to lock a whole table or a whole row of a table to protect the information in just one field. The longer data is locked and the more data is locked, the more impact the transaction can have on system performance, as one process after another stops to wait its turn to access the data.

However, with object-based transaction control, it is possible to minimize the performance impact of transaction control by locking information only when necessary and by locking only as much information as necessary. The ability to do this depends on the transaction models supported by your e-business system.

An application server with object-level transaction control can support both *optimistic* and *pessimistic* concurrency control. In optimistic concurrency control, objects can be read and written at will as if there were only one user, letting the OTM track changes and flag transaction conflicts at commit time. This approach requires no extra programming to set and release locks, and it can be highly performant because all users have free access to the same business objects. There is the risk that an application will not be able to commit a transaction, but for most Web-based applications which commit after each operation, there is little risk of multiple users working with the same critical data at the same time, and an occasional retry can be handled easily and cheaply. For the rare operations that cannot risk an inability to commit, the developer can use pessimistic concurrency control (locking). A good object store also supports very fine-grained locking, down to the level of a single object. This also improves performance, because the smaller the amount of locked data, the less chance multiple processes will need to access it at the same time.

Application servers without object transaction control support only pessimistic concurrency control. Because an RDBMS cannot track changes happening in the server, they can control transactions only through locking. To mitigate the performance loss as processes wait for data, some older application servers recommend two data paths for all applications data, one for data to be read and one for data to be written, so that locks are set only on writes. To create multiple access paths, developers have to do twice as much coding to use applications data.

Transaction Control and Entity Beans

In J2EE applications, the transaction control mechanisms offered by the application server can affect or control the design and performance of components such as Java entity beans. To make applications performant, quick-to-deploy and maintainable, developers need to be able to represent business objects as appropriate to the applications, not as required by limitations of the system architecture.

As we just discussed, in simple clustering architectures, clients must wait in line for access to the same data rather than working in parallel with their own transactional views. In such a system, this becomes the controlling factor in the design of the entity beans that represent business objects. The simplest and most intuitive way to design an application is to have course-grained entity beans that represent a single business object such as a product, which might factor out into numerous rows in numerous relational tables. This single object can then be the point of management and interaction for all data and methods related to that

product. In a sophisticated clustering system with a transactional object cache, this works fine because many threads in many VMs can work concurrently with their own transactional views of the product object and the OTM will handle any concurrency issues. For example, GemStone/J's transactional cache allows highly performant optimistic concurrency control or fine-grained locking with coarse-grained, easy-to-code entity beans.

However, in a simple clustering system, the only way to ensure synchronization is to have one copy of an entity bean in the single VM's cache and to have threads wait in line for access to that bean. To minimize this wait time or possible deadlock, developers are forced to create "fine-grained" entity beans: in other words, one entity bean for each row of a relational table. On this kind of server, a product bill of materials might take a bean for each sub-product, etc. Extra code is required to create and manage all these entity beans, and the overhead of creating the beans, doing garbage collection on them, and synchronizing them across a cluster can seriously degrade system performance. In fact, in its Java documentation, Sun Microsystems explicitly recommends against using this kind of fine-grained entity bean.

Managing Distributed Transactions

One of the greatest requirements of a powerful B2B application server is that it can provide concurrent users and applications with shared, transactional access to business information composed of data from multiple, heterogeneous databases. In our digital exchange example, a server with an OTM can handle two-phase commit and recovery for transactions across hundreds of supplier's systems.

A simple clustering architecture could achieve distributed transaction control by adding a traditional Transaction Processing Monitor (TPM). But TPMs are designed and optimized mainly for transaction control involving mainframe systems. Incorporating a TPM into an object-based system requires many developer-months of extra coding and integration, and because a TPM is not object-oriented, it will always be inefficient for object transactions. In fact, some application servers offer so called "enterprise" options which include a TPM, but these add-ons are an entirely different code base and work poorly with their Java server products.

Total Availability Technology

"Denial of service" is a phrase that strikes fear in the heart of every e-commerce professional. We all know that performance is irrelevant when a system is unavailable. Poor performance will alienate customers over time, but denial of service brings business to a stand-still. So real-time performance must be defined as performing all the time. To be truly performant, an e-business platform must achieve total availability.

We often confuse high availability with system failover. In simple clustering architectures, system-level failover is the main availability mechanism, hence the definition fits. However, a more mature multi-VM clustering architecture offers more efficient fail-over and higher overall availability:

- Highly stable Java VMs provide a first line of defense against failures.

- System-level failures are handled with enterprise cluster servers that provide high-performance fault detection and failover for the network, host platform and disk components of the system.
- Precision Failover ensures continuous service for software components within a system.
- Dynamic system management allows system expansion, reconfiguration and tuning, application deployment and updates while the system is on line.

Stability of VMs

Java VMs do the work in a J2EE system, so the first and best way to ensure high availability is to choose an application server with extremely stable VMs. In looking for VM stability, look for a platform vendor with extensive experience in distributed programming and VM architectures, and who has rich test suites developed for complex B2B environments.

System-level Failover

On the system level, an enterprise cluster server can monitor the network, system platform and disk using a dedicated "network" to send continual heartbeat messages between the protected machines in the cluster. Configurable policies should be available to dynamically determine where recovery occurs for the most efficient failover. And cascading automatic recovery should be available to handle consecutive failures.

Precision Failover

In modern computing environments, total machine failure most often happens due to natural disasters or major human error. (We recently heard an amusing story about an administrator who brought down a system by putting a the wrong kind of disk drive into a hot-swappable disk drive bay. Of course, the users of the system were not amused.) With today's robust host platforms, failure of lower-level computing resources is more of an on-going issue for e-commerce systems. For example, a VM might become unresponsive due to poorly performing application or third-party code.

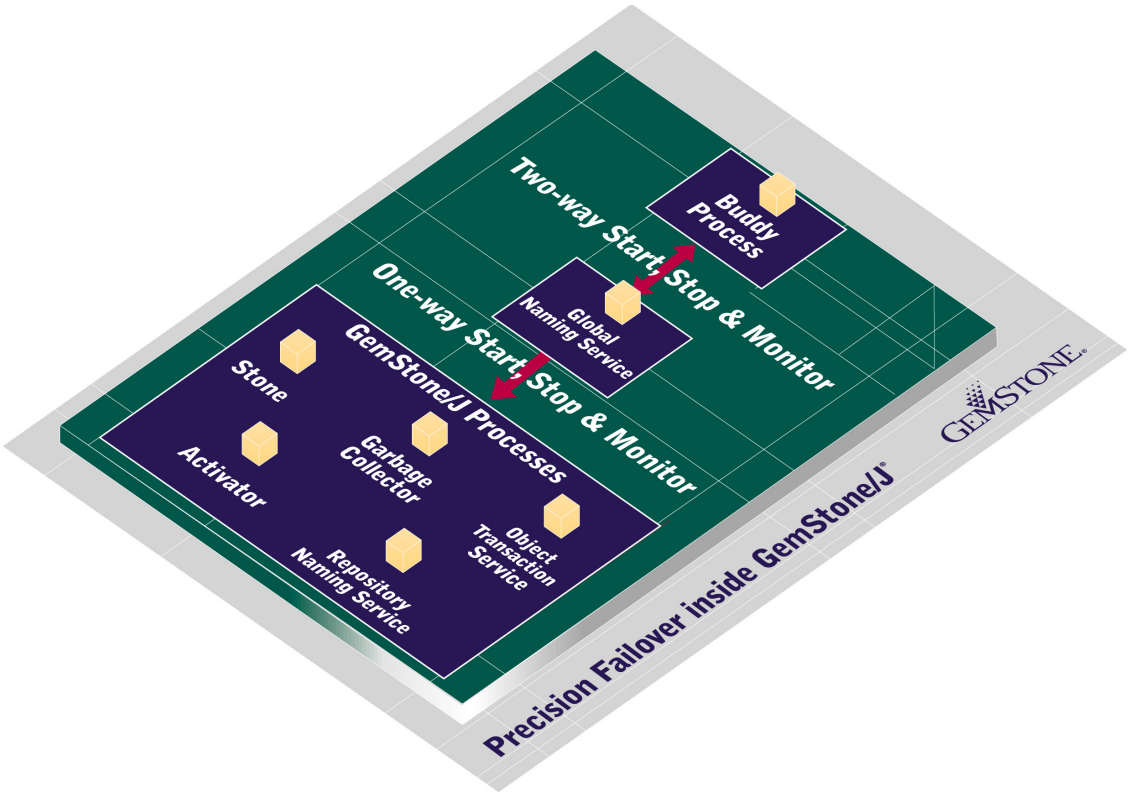

Figure 6. Precision Failover of Software Processes in GemStone/J.

Precision Failover technology monitors and handles recovery for critical software components and reinitiates the processes as necessary (see Figure 6). In GemStone/J, for example, software component failover is handled on numerous levels:

- The active Global Name Service (GNS) process monitors the other processes running in the system, including the Activator, Repository Name Service (RNS) and the PCA manager process. If any of these processes become unresponsive, the GNS will stop and restart it. The active GNS in the primary server also tracks the GemStone/J processes in secondary machines. A "buddy" process monitors the active GNS. In the event of a GNS failure, the buddy will shut it down and initiate a restart. Since many of these are separate processes, a failure of a component may not interrupt active requests in the system.
- If a VM fails, the Activator will automatically reroute client requests to another pooled VM or start a new VM if necessary. If the replacement VM is on the same machine, it may be able to continue processing using objects already in the shared object cache.
- Client session state is automatically stored in the persistent object cache, so if Servlet engine VM fails, the Web Server adapter can reroute requests to another Servlet engine VM, which then retrieves session state from PCA using the client ID.

Dynamic System Management

A key element of high availability is the ability to change system configuration and application software without taking the application server down. An e-commerce platform must be able to scale and adapt while remaining on line, whether scalability is enhanced

through addition of new VMs or other software components, or by reconfiguring for performance tuning.

Because they scale primarily by adding new hardware, simple clustering systems are impractical for a high-end production site that requires very high availability. A multi-VM architecture can be scaled without taking systems down and, often, without adding new systems. Sophisticated clustering systems can dynamically add, remove, and reconfigure VMs as load on a site increases while remaining on line, whereas a simple clustering system must add new hosts, therefore new IP addresses, and then reboot every time a new VM is added.

Comprehensive APIs are also important to provide dynamic control of all parts of a system. These tools should include both command line controls and a graphical UI, and configuration changes should be saved automatically. Some older application server architectures lack these capabilities. Any configuration changes made while a server is on line are lost on restarts. To make lasting configuration changes on these systems, you have to shut down the server, edit the properties file, and restart the server with the updates.

Scalability for the Next Incarnation of B2B Commerce

Day to day, e-business systems require scalability to handle increasing numbers of users in a performant way. In the longer term, scalability is about the ability to integrate new processes and partners and to take advantage of new opportunities.

Consider our digital exchange company today and tomorrow. Already this application combines several architectures into one application or set of components: EJBs, CORBA objects, JSPs, and servlets. A common, well-integrated environment is crucial for success. But there are more integration challenges ahead. Today their system does daily batch loads from affiliate catalogs; when prices change between one day and the next, they absorb cost differences. Tomorrow they'll integrate with supplier systems to update prices in real time. Today their customers hand enter their materials lists for large projects. Tomorrow customers will load lists from their own systems and keep their own document "file cabinets" on line.

Integration like this calls for an application server that supports a diverse set of applications and requirements. Such a server must be designed for heterogeneous business environments:

- It must be Java-centric, not just EJB-centric.
- Business objects can be written in pure Java and accessed through EJBs, but they can also be accessed through CORBA wrappers, through distributed Java beans, RMI or other ways.

With this kind of open architecture, a next-generation application server is easy to integrate with other business systems, leveraging its highly scalable, highly available architecture to automate and support even more diverse business processes.

Start Now, Start Smart, Stay Ahead

Every business magazine, every trade journal, every new word that appears in print with an "e-" in front of it, trumpets the fact that e-commerce is an imperative. A short time ago, having an e-business strategy was a differentiator. Now it's a necessity to stay in the game.

Differentiation, especially for B2B e-commerce, comes from the ability to efficiently integrate and automate complex business processes in partnership with your partners and customers. The message is to start now, start smart, and stay ahead.

While first-generation clustering application servers may support a company's first steps into e-commerce, they will not scale to support growth, either in terms of performance or business automation and integration capabilities. In choosing an e-commerce platform, keep these requirements in mind.

- Look for extremely scalable technology that includes multi-VM architecture, object-based transaction control, and dynamic system management.

- Scalability starts with technology, but it is realized through design. So find a vendor with a proven track record of designing scalable systems. There are very few vendors with the years of experience and know-how to design performant distributed business systems.

- In planning projects "tactical development of strategic architecture." In other words, you can start slow with e-business initiatives, but know where you're going. Build what you need today on a foundation that will take you anywhere you need to go in the future.

With the right platform, the right development partners, and a clear vision, you can expand your e-business capabilities and results smoothly from the first dynamic Web site to total enterprise automation.

About GemStone Systems, Inc.

GemStone Systems, Inc., a Brokat Company, is the software infrastructure technology leader for the new B2B economy. GemStone's Java Success technology initiative provides e-businesses with faster time-to-market and enhanced competitive advantage for their Internet marketplaces. The GemStone e-business software solution incorporates industry standards in the most advanced, integrated architecture of any product in its class. Today, customers are using GemStone's Extreme Clustering™ technology to deploy production B2B environments in all walks of the dot-com world. GemStone's Java-based platforms are customer-tested and proven scalable, secure, adaptable, and reliable.

Headquartered in Beaverton, Oregon, GemStone distributes and services its products worldwide through offices in the United States, UK, France, Germany, Sweden, and Switzerland, as well as through a network of distributors. For detailed product and company information, please visit www.gemstone.com.

About Brokat AG

Brokat AG is among the world's leading providers of software for e-Business solutions. Its key product is the modular e-Services platform Twister, a multi-industry software that integrates existing IT systems and applications in companies and securely links to various electronic channels, such as the Internet and mobile communications. Brokat was founded in 1994 and currently employs more than 800 employees in 16 countries. In calendar year 1999, the company posted revenues of DM 94 million. Brokat has its headquarters in Stuttgart.