

OSS THROUGH JAVA™ INITIATIVE

A new breed of OSS solutions

Deployment Strategies focusing on Massive Scalability

Written by: Brian Naughton, Sonic Software
Brian.Naughton@sonicsoftware.com
Massive Scalability Focus Group

Document Version: 1.0
Date: 25 April 2003

Edited by Brian Naughton, Sonic Software

Contributors:

Scott Cranton, Progress Software

Chris King, BEA

Nagendra Nagarajaya, Sun Microsystems

Thomas Schmal, IP VALUE

Dirk Vleugels, IP VALUE

Dave Winstone, Sonic Software

Table of Contents

TABLE OF CONTENTS	3
EXECUTIVE SUMMARY	5
1 INTRODUCTION	6
1.1 SCOPE	7
2 BUSINESS AND APPLICATION ENVIRONMENT	8
2.1 USE CASES	9
3 APPLICATION DESIGN	10
3.1 HIGH LEVEL DESIGN	10
3.2 LOGICAL ARCHITECTURE	11
4 PATTERNS AND STRATEGIES	13
4.1 DEPLOYMENT PATTERN TEMPLATE	13
4.2 WEB SERVERS.....	14
4.2.1 <i>HTTP Load Balancing</i>	14
4.3 SERVLET CLUSTER	15
4.3.1 <i>Application Server Clustering and Pooling</i>	15
4.3.2 <i>Transactional Data Caching</i>	16
4.3.3 <i>Cache Routing and Affinity</i>	18
4.4 DATA CLUSTER	20
4.4.1 <i>MDB Pooling</i>	20
4.4.2 <i>Data Partitioning</i>	21
4.5 MESSAGE BROKER CLUSTER.....	23
4.5.1 <i>Broker Clustering and Routing</i>	23
4.6 SERVICE ACTIVATION	24
4.6.1 <i>Hierarchical Distribution</i>	24
5 TEST METHODOLOGY	26
5.1 PERFORMANCE MEASUREMENT TOOLS - THE GRINDER	26
5.1.1 <i>Where to obtain the Grinder</i>	27
5.2 PERFORMANCE METRICS DEFINITION	27
5.2.1 <i>Customer Status Requests</i>	27
5.2.2 <i>Orders Requests</i>	29
6 RESULTS DISCUSSION	31
6.1 END TO END USE CASE	31
6.1.1 <i>Configuration</i>	31
6.1.2 <i>Running the tests</i>	33
6.2 SINGLE CASE.....	36
6.2.1 <i>Configuration</i>	37
6.2.2 <i>Running the tests</i>	38
6.3 TCP TUNING.....	41
6.3.1 <i>System tuning</i>	42
7 CONCLUSION	43
APPENDIX A – HARDWARE AND SOFTWARE DETAILS	45
HARDWARE	45
SOFTWARE	45

REFERENCES 46

Executive Summary

Over the last decade OSS developers have moved away from monolithic, telecommunications-specific architectures and embraced off-the-shelf multi-tier architectures and distributed-object middleware. Over the same period, equipment-vendor-only solutions have given way to multi-vendor solutions that include products from Independent Software Vendors (ISVs), System Integrators (SIs) and middleware vendors.

Initiatives such as the OSS through Java™ Initiative (OSS/J) promote the adoption of a component-based approach to developing OSS solutions, by kick-starting a component marketplace that can ultimately offer interchangeable, interoperable components that can be rapidly assembled into OSS solutions. To be successful though, it is imperative that the OSS/J base technology choices can provide the carrier-grade performance and scalability required in today's OSS environment.

This paper demonstrates OSS/J deployment strategies and patterns for the introduction of a typical service provider use case that manipulates a number of OSS applications in a manner outside their original design criteria. The strategies and patterns detail how the OSS/J integration infrastructure can accommodate any performance or scalability shortfalls that may exist between the back end OSS applications and the introduced use case requirements.

As the basis for the Massive Scalability use case, a customer self-service portal that opens a number of back office OSS applications directly to the service provider's customer base was chosen. This use case places new performance and scalability requirements on the back end OSS systems that would certainly not have been anticipated at the time of their design.

Off the shelf standards based IT integration products and applications, available on the market today, combined with a commodity hardware configuration, produced the OSS/J deployment environment. The demonstration system scaled to handle over 16.2 million customer requests per hour. ***Handling 16.2 million requests per hour equates to the entire customer base of a large service provider actively requesting real-time information concurrently within the same hour.***

Scaling the system was a matter of configuring the OSS/J deployment environment as necessary rather than rebuilding or re-customizing the OSS applications themselves every time the load requirements or use cases changed. This means as a service provider evolves its operational environment it can do so without dramatic disruption to its OSS application assets.

The distribution techniques used allow the system to be scaled by adding additional commodity hardware as needed, taking an incremental approach to scaling the system rather than needing to retrofit the hardware platform with more physical resources such as processors, memory or disks etc.

System performance and capacity overhead/degradation as a result of employing these distribution patterns was shown to be predictable enough to be suitable for planning purposes. This means of course that a service provider can now plan with a certain degree of accuracy the environment that would be needed to support certain OSS performance requirements. Planning operational budgets can now become more of a science than a fine art.

1 Introduction

Now that the OSS/J has completed its initial scope of work, and verified and demonstrated the OSS/J technology and architecture choices, the next stage of the initiative broadens its focus to include other application areas of OSS and also include practical deployment aspects of OSS/J based solutions.

As the OSS/J moves from specification to adoption, tried and tested deployment strategies and implementation/integration best practices are key to the success of the initiative. To address these deployment issues the OSS/J has created a number of technology focus areas that are tasked with producing tried and tested best practices and deployment strategies in the context of a typical complex OSS/J deployment. One of the key deployment issues that needs to be addressed is that of performance and scalability, it is crucial that organizations implementing OSS/J solutions have the necessary support with respect to implementation and integration strategies to successfully meet the performance and scalability requirements of a typical service provider environment.

This white paper details the results of the first phase of the “Massive Scalability” technology group whose objective is to produce best practices and deployment strategies that address scalability issues horizontally across the OSS/J. In keeping with the philosophy of the OSS/J these documented strategies and best practices must be proven in a realistic OSS environment to meet valid performance and scalability requirements. Therefore rather than providing an academic set of strategies and recommendations, this paper is driven by an extensive implementation/testing effort and the experiences gained that is designed to validate any produced work.

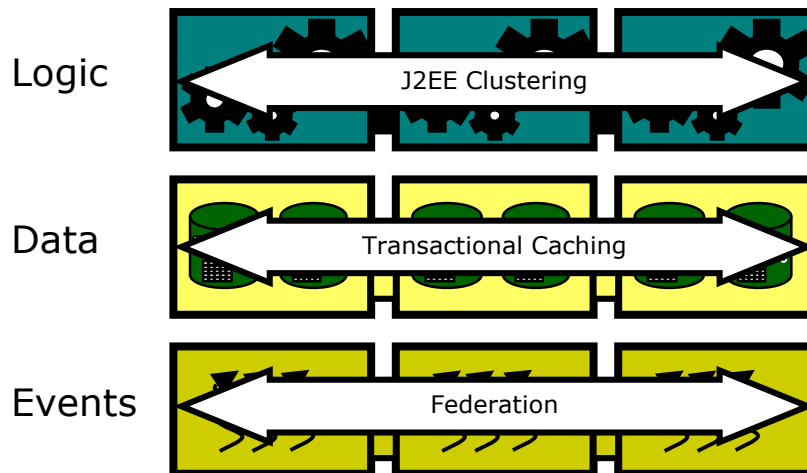


Figure 1 Scalability Patterns

Documented best practices and deployment strategies take the form of design patterns and implementation recommendations that address scalability issues across a number of architectural areas that are common to OSS deployments (see Figure 1 above), namely:

- *Scaling of logic*: Serving very large numbers of concurrent clients within an adequate response time
- *Scaling of events*: Managing large numbers of events, e.g. billing, performance metrics, faults without compromise

- *Scaling of data*: Integrating many back end systems without compromising performance and scalability

The above design patterns and implementation strategies are applied in the context of existing OSS/J functional APIs, e.g. Service Activation, Trouble Ticketing etc. Typical OSS use cases are built out and demonstrated that address the three architectural areas described above. The tailored patterns and recommendations are implemented and then tested and evolved to meet the desired requirements. The implementation and deployment experiences gained with any considered pattern or strategy is fully documented to give a full account of each pattern's benefits and pitfalls.

Deployment patterns and strategies are also designed to take advantage of products and standards that are available today, giving organizations a realistic understanding of how the patterns and strategies can be realized to achieve predictable scalability given their customer environments and requirements.

1.1 Scope

Produced deployment strategies and best practices concentrate on addressing scalability and performance issues within a typical OSS/J environment only. Issues such as high availability while supported by the techniques described within this paper are outside the scope of this work.

Strategies and patterns are based on the J2EE programming model, Web Services, XML, and products and standards that exist on the market at the time of writing this paper.

2 Business and Application Environment

Service providers are aggressively using the web to allow customers to serve themselves, e.g. check order or trouble ticket status, specify product configuration and view billing information. For a typical tier 1 service provider this means opening existing OSS applications to millions of customers. This is typical of self-service, b2b, service on demand type strategies, where integrated OSS applications are opened up to unpredictable use cases that they have never been designed to accommodate. In these cases it is up to the integration infrastructure to scale to meet these new demands.

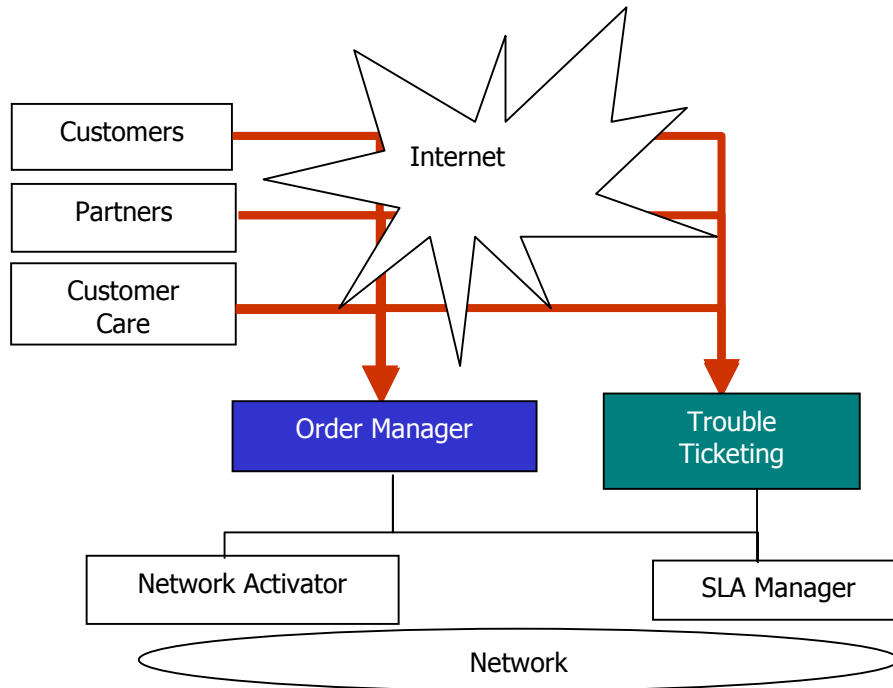


Figure 2 Business & Application Environment

For this iteration of the massive scalability use case we introduce a new customer self service touch point into a service providers operations that needs real-time access to the status of customer orders and trouble tickets. This self-service portal application opens up a number of OSS functions directly to a service provider's customer community, that is all functions within the organization that are responsible for interacting with customers as well as interacting directly with the customers themselves.

The customer portal in this case exposes *Service Activation and Trouble Ticketing* functions to the customer community, allowing them to directly create orders and trouble tickets and query for the status of any existing orders or open trouble tickets. The business problem describes mission critical, 7x24 scenarios that require extreme performance and scalability.

Figure 2 depicts the OSS environment for the chosen business problem. Service providers need to permit customers and partners direct access to service configuration, ordering, and status checking. Also any faults in the service provider network assets that affect the delivery of these ordered services must be acknowledged and resolved to avoid any customer or partner service level agreement penalties.

The environment above was deemed to meet the requirements for this iteration of the massive scalability effort for a number of different reasons:

- Performance requirements must meet customer conversational speed expectations.
- Integration infrastructure must accommodate the necessary performance requirements to boost the performance of the individual OSS products.
- Integration infrastructure must accommodate functionality changes over time without disruption or compromising performance and scalability.
- The scenario must be capable of scaling as the customer base grows.

To make the scenario as realistic as possible, we must accommodate a number of different use cases that will stress the system under test to mimic a realistic service provider deployment.

Therefore the following requirements have also been placed on the system:

- A constant stream of orders must be processed at all costs. That is a service provider needs to know that it can provision subscribers at a predictable rate and with a predictable response time.
- The systems must also accommodate a spike in activity without affecting the constant stream of orders. Trouble tickets raised through violations in the network will affect a subset of customers, affecting either orders in progress or services that are already provisioned. This has the affect of aggressively increasing the number of affected customer queries on trouble ticket or order status. Therefore introducing an unpredictable spike of activity into both systems.

The system must accommodate the variety of ways in which a service provider can grow. These include simple growth as well as diversification and corporate mergers. The first method, growth, means simply that the service provider gains more customers and partners and sells more products and services. The networks grow larger, and it has more of them.

The company can also grow via diversification or through mergers. In both these cases, the company, now a conglomerate, may take on entirely new lines of business. It sets up new divisions, each with its possibly distinct set of customers and suppliers.

2.1 Use Cases

In this context the most business critical operations are:

- *Order and Trouble Ticket Status:* The status of customer orders and trouble tickets is made available to all customer functions, i.e. call center, self service portal, B2B gateways. The status must be received within a conversational timeframe in order to match performance expectations.
- *Order Creation:* A steady stream of orders must be processed daily irrespective of any other distractions to the order system.

The number of orders processed by a service provider in a given time period is quite predictable and unlikely to dramatically change based on the services they offer and the size of their existing customer base. Therefore system requirements that can meet the performance and scalability requirements of this particular process can be based on the growth of the customers and services offered.

However status requests from customer functions are not as predictable in frequency or numbers as orders, which means that the ordering process must also be able to cope with infrequent and unpredictable amounts of spiked requests and just as critically ensure that the ordering process still performs within acceptable boundaries.

3 Application Design

This section describes the architecture for our Massive Scalability use case. Detailing the various architectural layers and components within the solution and the high level patterns that are employed to scale them.

3.1 High Level Design

The demonstration architecture breaks our use case into two separate parts, effectively dealing with customer read and write requests to the system separately:

- *Write requests* such as create or modify orders or trouble tickets are passed directly to the back end OSS systems that own those functions
- *Read Requests* such as order and trouble status queries, are taken away from the back end OSS systems and instead managed by a customer shared information staging area that represents an up to date and common view of the requesting customer.

In this way the read and write requests can be scaled independently. Write requests are scaled by a variety of distribution techniques, for example partitioning order-processing logic into sub systems that are responsible for geographic areas, then routing and load balancing across the entire hierarchy.

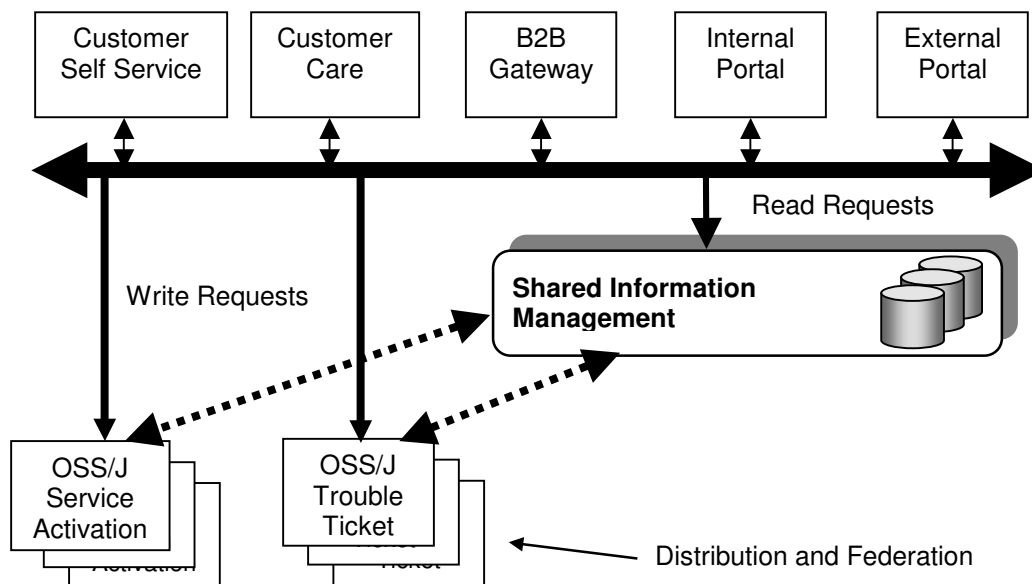


Figure 3 High Level Architecture

The customer shared information store reconciles with the back end OSS systems to keep a complete up to date copy of what a customer looks like, and as such is completely decoupled from the performance characteristics of the back end OSS systems. Status requests can then be scaled separate to the scaling of create or write requests. This has the effect of relieving the back end OSS from a lot of the unpredictable nature of the customer behavior. Customer read interactions could be responsible for up to 90% of customer's interactions so splitting the request

handlers in this fashion allows the write case to get on with predictably processing orders or trouble tickets.

3.2 Logical Architecture

The deployment architecture consists of a number of tiers, each responsible for a specific logical function within the overall solution. Each tier is a logical partition of the separation of concerns of the system. Each tier is assigned its unique responsibility in the system. We view each tier as logically separated from one another. Each tier is loosely coupled with the adjacent tier. We represent the whole system as a stack of tiers. Each tier can be scaled independently of the others. As seen in figure 4 below the demonstration consists of the following tiers:

- **Web Servers:** Accepts customer HTTP read-only status requests and passes to the servlet cluster layer.
- **Servlet Cluster:** Responds to HTTP requests from the web servers with the appropriate customer information from the underlying data cluster.
- **Data Cluster:** Provides a unified view of all customers and their related order status.
- **Message Broker Cluster:** Provides the infrastructure for managing the propagation of events from the back end operational systems to the data cluster.
- **Service Activation:** The back end OSS systems that own the customer information.

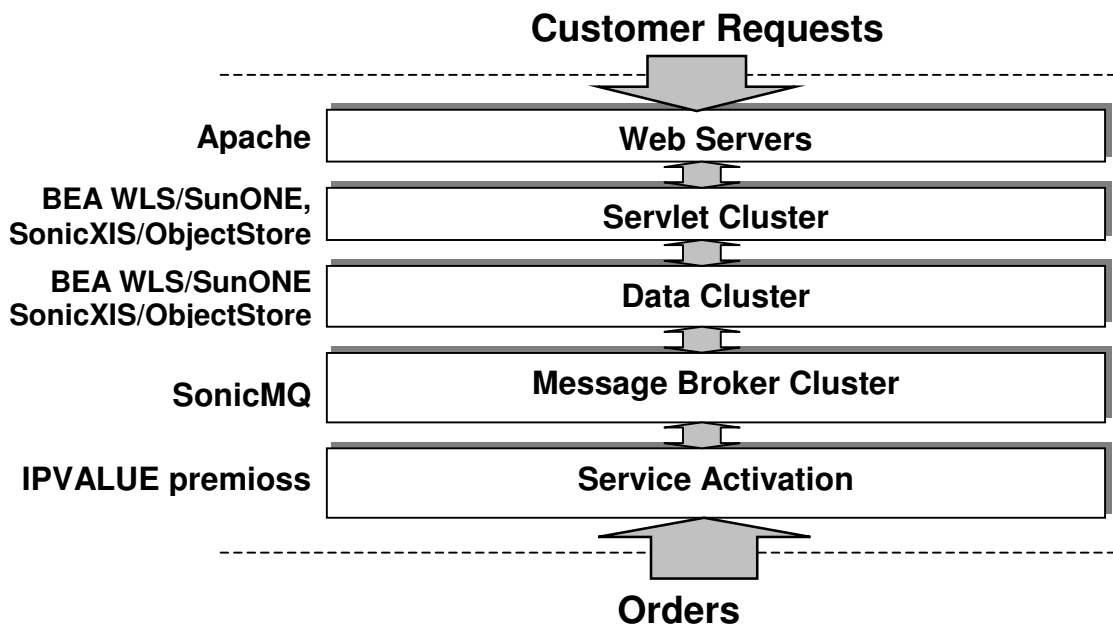


Figure 4 Multi-tiered System

Each individual tier can be scaled through a number of deployment patterns independently from the other tiers, i.e. with little or no disruption to the other tiers. Also each tier and its associated deployment patterns and strategies are based on features within off the shelf products that exist on the market today. In this way as we encounter bottlenecks or performance issues, the tier that needs to be scaled can be reconfigured as opposed to having to (re)write application code to resolve the problem.

Each layer communicates to the layers above and below through standard communication channels and API's, whether OSS/J based API's for communicating with the OSS business components or J2EE and XML based API's for messaging, data management etc. This means

that each layer can be decoupled from the others allowing the products within the layer to be configured or replaced without dramatic changes to the system.

As shown in Figure 4, customer status requests come in to the web server layer and are routed to the servlet cluster below. The servlet cluster responds to the customer requests with an up to date unified view of the customer, supported by the underlying data cluster. These tiers are decoupled from the service activation and trouble ticketing systems through a JMS tier that provides an asynchronous communication mechanism that decouples the performance characteristics of the back end system from the tiers above.

It is imperative that all of the tiers that respond to customer requests can be scaled rapidly to respond to unpredictable customer traffic causing surges and spikes in requests, and also that the underlying business systems are unaffected by this new customer load.

The back end orders are batched straight into the service activation system, and any order state changes are then propagated automatically via the OSS/J API's notification mechanism over JMS. In this way the customer shared information store is kept in sync with the back end service activation system.

The next chapter goes into each tier in more detail, describing the deployment patterns and referencing the products used for the demonstration.

4 Patterns and Strategies

This section details the deployment patterns and implementation strategies used throughout the Massive Scalability effort. These deployment patterns and implementation strategies are designed to promote performance and scalability in a typical OSS/J integrated environment.

A deployment pattern defines a recurring solution to a problem in a particular context. A context is the environment, surroundings, situation, or interrelated conditions within which something exists. A problem is an unsettled question, something that needs to be investigated and solved. A problem can be specified by a set of causes and effects. Typically, the problem is constrained by the context in which it occurs. Finally, the solution refers to the answer to the problem in a context that helps resolve the issues.

The deployment patterns should communicate design solutions to developers and architects who read and use them. In our catalog, a pattern is described according to its main characteristics: *context*, *problem*, and *solution*.

A pattern describes, at some level of abstraction, an expert solution to a problem. Our patterns are documented in a template form. Patterns document solutions that promote performance and scalability at every layer within our architecture. The documented deployment patterns are defined at a component level of granularity and are based on the experience of the vendors involved and the combined capabilities of the involved vendor products.

At the same time, each deployment pattern includes various implementation strategies that provide lower-level implementation details. Through the strategies, each pattern documents a solution at multiple levels of abstraction.

4.1 Deployment Pattern Template

The massive scalability deployment patterns are all structured according to a defined pattern template. The pattern template sections each contribute to understanding the particular pattern and are based on *Sun's Java Center J2EE Patterns* [1].

The pattern template consists of the following sections:

- **Context:**
Sets the environment under which the pattern exists.
- **Problem:**
Describes the deployment issues faced.
- **Solution:**
Describes the solution approach briefly and the solution elements in detail. The solution section contains the following sub-section:
 - **Strategies:**
Describes how the pattern is implemented. What products were used etc.
- **Consequences:**
Here we describe the pattern trade-offs. Generally, this section focuses on the results of using a particular pattern or its strategy, and notes the pros and cons that may result from the application of the pattern.

4.2 Web Servers

The web server layer is responsible for managing all interaction with client devices or systems. A client could be a web browser, an application, a phone, or a network application. In the context of our use case a client is any device that either a customer or internal service providers departments on behalf of the customer can communicate through.

4.2.1 HTTP Load Balancing

Context

Large volumes of client HTTP requests are routed through the web server cluster to the available servlets underneath to fulfill those requests.

Problem

At this level the problem is how to efficiently manage the huge number of client requests and route them to the available underlying servlet instances to fully utilize the hardware and software resources.

Solution

Load balancing HTTP requests allows the most efficient concurrent use of the available resources underneath. As customer requests come in they are routed to the next available server allowing an even spread of requests across all available servlet machines beneath.

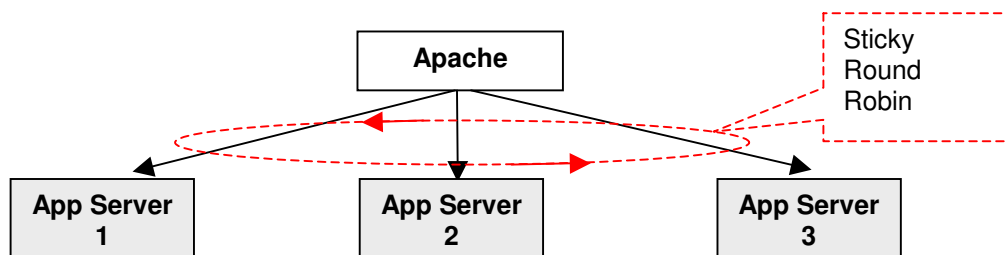


Figure 5 HTTP Load Balancing

Strategies

We used apache with a BEA plug-in to load balance HTTP requests across a cluster of WebLogic servers. The apache plug-in performed rudimentary sticky round robin load balancing, which was more than adequate for our purposes.

A number of alternative methods could also have been used to perform this task. The most efficient method would obviously be to replace Apache with hardware based IP load balancers, or also specialized IP load balancing software could have been used.

Consequences

Efficient utilization of available hardware and software resources.

4.3 Servlet Cluster

This tier encapsulates all presentation logic required to service the clients that access the system. The servlet cluster accesses the customer shared information, constructs the appropriate response, and delivers the response back to the web server cluster above. Servlets and JSPs are the mechanism for delivering this logic.

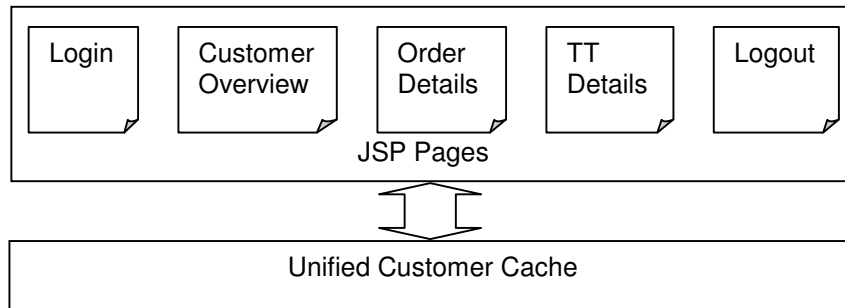


Figure 6 Servlet Architecture

Our Servlets consist of a number of JSPs and Servlets that access customer information from a shared data cache. Servlets are pooled within an application server, and each application server has a single instance of the shared information cache, pooling servlet requests to the caches also.

A number of patterns are used to scale the servlets:

- Application server pooling and clustering
- Transactional Data Caching
- Cache routing

The following sections go into more detail for each of these patterns

4.3.1 Application Server Clustering and Pooling

Context

The web servers pass huge volumes of customer HTTP requests to the servlet cluster.

Problem

The Servlets and JSPs that hold the presentation logic need to respond to massive volumes of client requests in as short a timeframe as possible. The Servlets and JSPs need to efficiently handle concurrent requests from thousands of clients in an acceptable response time. Also the deployed Servlets and JSPs need to be able to scale to meet future requirements.

Solution

Out of the box J2EE application server features that promote concurrent and distributed access to our servlet logic are important patterns for promoting scalability at this level. Our servlet logic is packaged and deployed to an application server and the application server is then configured and tuned to get the best results.

The J2EE application server makes available two very important features to manage the above

- **Connection and resource pooling management**, efficiently utilizing the application server and system resources and promoting concurrent and parallel processing to avoid system or processing contention issues.
- **Clustering** of distributed application server instances to scale beyond the limits of a single machine.

Servlets can be associated with Servlet Pools within the application server and the administrator can then tune at run time the characteristics of the servlet pool, for example the maximum number of servlet instances or the initial number of instances at startup. In this way we can rapidly tune the system for the best performance.

Another interesting tuning point with respect to the servlet pools are the number of system threads available to service the tcp pools and also the system requests. This can have an affect on performance.

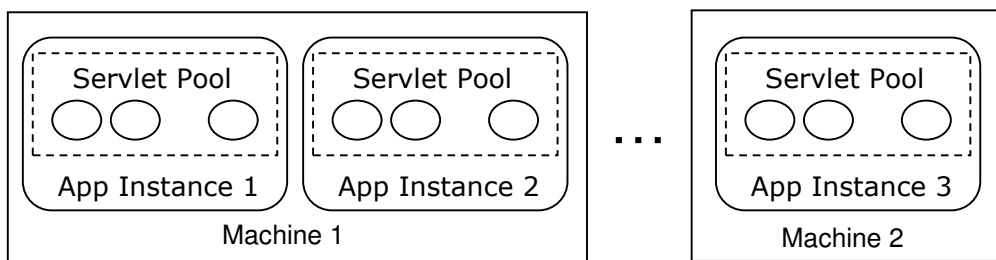


Figure 7 Application Server pooling and clustering

When we reach the limits of a single instance of an application server or indeed the physical limits of the hardware we need to spread the load across a number of machines to scale the system further. Application server clustering allows us to seamlessly distribute application server instances across machines and maintain a single logical entity to respond to the customer requests.

Strategies

Both BEA WebLogic Server and SunOne AS 7 provided servlet pooling out of the box. We used BEA WebLogic server in clustering situations across machines, see the Results Discussion later for more detail.

Consequences

Pooling and distribution are administrative tasks that allow the tuning of a distributed system to be a rapid one. Discovering which patterns and tuning fit a particular use case is a pretty quick and relatively easy task, i.e. there is no need to re-develop any application code.

4.3.2 Transactional Data Caching

Context

The Servlets and JSPs provide direct access to customer information that can change from one client request to the next as data changes in the back end service activation system. This means

each and every request must go straight against the underlying Data Cluster to ensure consistency.

Problem

There are potentially dramatic contention issues allowing massive numbers of requests to access the same physical customer storage mechanism (e.g. the same physical disk or database system). This would introduce a bottleneck to the system that would limit performance and scalability to all other layers within the system.

As well as the contention issues that surround the retrieval of data from system hardware, funneling all customer traffic to a single point within the system will also flood the system network causing another potentially dramatic bottleneck.

Solution

To relieve this bottleneck we cache the customer data out at the application server instances where the information is needed. This has the effect of reducing network traffic and allowing the servlets access to customer data at in-memory data speeds, rather than sending requests over the network to a shared disc each and every request. The issue now of course is how to maintain transactional consistency across the distributed customer caches.

The cache is replicated across all app server instances and kept in sync typically through replication or lazy propagation techniques.

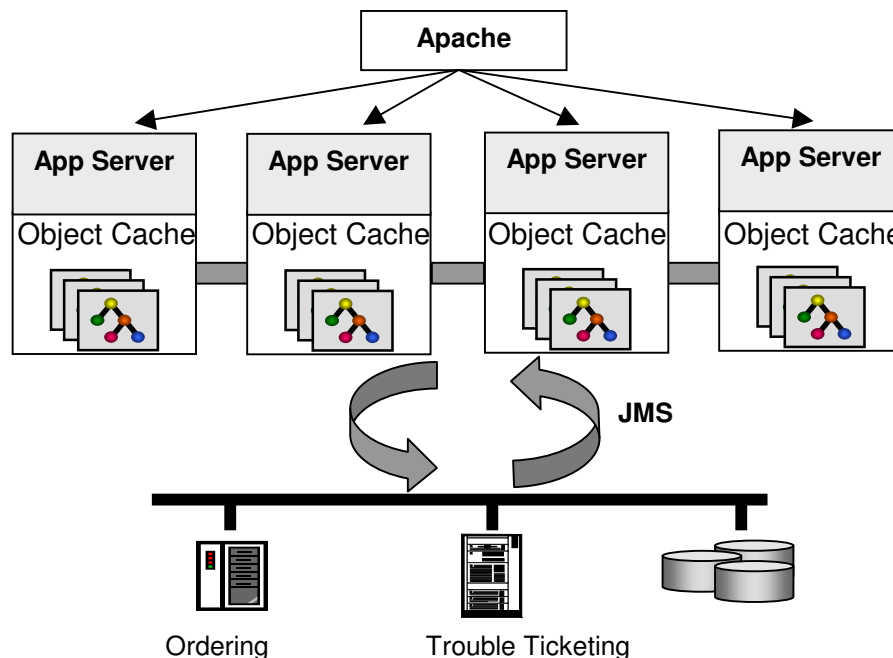


Figure 8 Transactional Caching

Only delta changes are sent across the network when the new customer information is requested, therefore removing the need to broadcast changes as they are propagated from the back end service activation system.

Strategies

There are a number of data management products on the market that specialize in this area. We interchangeably used two products that provide transactional caching. From Sonic Software we used SonicXIS and from Progress we used ObjectStore. Both products offered transactional caching.

Consequences

Provides huge performance gains at the servlet cluster with in-memory customer data access speeds. The write changes from the back end ordering systems are infrequent compared with the throughput of the customer reads, so keeping transaction consistency across all application servers is not system intensive.

4.3.3 Cache Routing and Affinity

Context

This pattern is an evolution of the previous pattern (4.3.2) to handle very large data sets. Tier 1 Service Providers will have a customer base with tens of millions of customers. This in turn means that any “common” information tier must be able to accommodate high performance and scalable access to something in the order of 100s GB of customer data.

Problem

The pattern described in the last section is not efficient if the customer data set is larger than the available memory on each of the application server machines. Therefore a more intelligent caching strategy is necessary.

Solution

In the case where all customer data is cached on every application server and the data set is much larger than the available memory there can be a lot of system time spent mapping customer data into the cache and evicting data out of the cache. This is counter-productive and defeats the purpose of caching to a certain degree, with in-memory speeds not achievable with a random spread of customers requests (due to the constant eviction and loading of data).

Rather than caching all customer information at every instance of an application server this time we introduce an additional layer to the servlet cluster that routes data requests to application servers that are responsible for a segment of the customer data. Typically responsibility for data segments is based on geography or a similar partitioning strategy, but ultimately the data segment will be partitioned into chunks that can fit completely into the available memory for each application server instance.

Partitioning sets of customer information into groups of application servers and then routing requests allows us to extend the customer information model dynamically by simply extending the routing algorithm to include a new application server that holds a new set of customers. A new dataset can be brought on dynamically without disrupting the others, also they can be evolved one at a time etc. and they can also be load balanced in pairs or triplets etc.

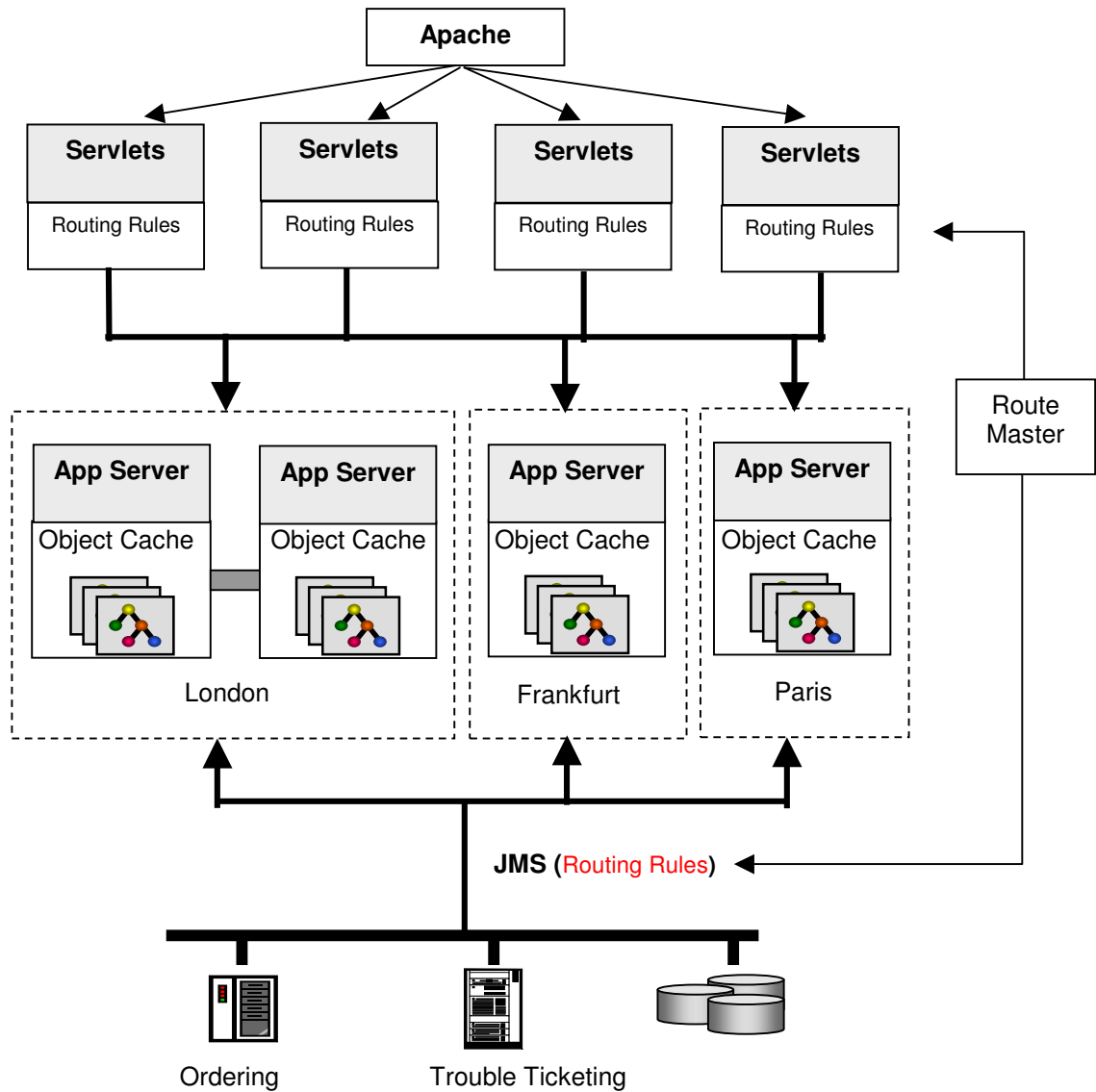


Figure 9 Cache Routing

Figure 9 above shows this pattern in practice, a global set of routing rules are defined that assign specific data sets to particular application server instances. In the case above, customer data is segmented by geography therefore London, Frankfurt and Paris customers are cached on separate application server instances.

More than one application server can represent the same customer data segment, effectively applying the previous pattern to a customer data segment. This means that we can now bring on new application servers for a particularly busy geography and then share the load.

Strategies

BEA’s WebLogic Server provided the necessary application level clustering support, allowing us to distribute logic across physical machines. SonicXIS and ObjectStore were used underneath the application server instances to manage the customer caches.

Consequences

This pattern is designed to allow the system to scale at every architectural point without the need to bring the entire system down. New machines with additional application servers and caches representing additional data sets can be added easily at run time.

4.4 Data Cluster

The data cluster is responsible for physically managing and storing all of the information that supports the servlet caches described earlier. Customer unified information is transformed from the back end service activation and trouble ticket systems and transformed into a format that is pertinent for the self-service portal use case. In this way pre-processing the customer information into a format that is friendly and relieving the servlet cluster from this type of transformation load.

This physical data store is updated through OSS/J's built in event propagation mechanism, where changes made in the back end OSS/J compliant applications are automatically published to a JMS topic for interested applications to do with what they please. In this way any changes in orders within the service activation system are associated with customers in the unified store.

4.4.1 MDB Pooling

Context

Changes from the back end systems are propagated to the physical customer data store. The OSS/J API's have a built in notification mechanism that allows applications to subscribe to changes that are interesting to them. In our case any changes to customer orders, and trouble tickets are propagated to the customer data store.

Problem

Efficiently manage and process the back end orders and update the affected customer in the customer data store.

Solution

Similar to the servlet pooling pattern described in section 4.3.1 the J2EE application server pooling and clustering capabilities allow us to efficiently manage the deployment of the Message Driven Beans that capture OSS/J order state change events.

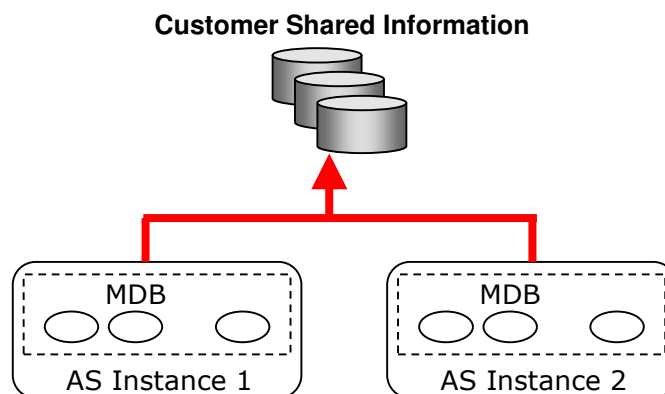


Figure 10 MDB Pooling

Strategies

Both application server implementations from both BEA and Sun provided the necessary MDB pooling out of the box.

Our MDB's interact with the customer shared information through the API's exposed by SonicXIS and ObjectStore, both products providing standard based API's. SonicXIS with standard XPath queries and Document Object Model API's and ObjectStore providing a Java Data Objects interface (JSR 12).

Consequences

Efficient asynchronous event capture.

4.4.2 Data Partitioning

Context

Message Driven Beans take OSS/J order state changes and update the physical customer data stores to reflect the new changes. The updated information is then immediately available to the customers themselves.

Problem

The physical customer store is another potentially huge bottleneck. If all customers are physically stored in the same disk set, then there is a single point of contention that could have scalability and performance implications on the rest of the system.

Also there are performance issues around the indexing mechanism used to access a unique instance of a customer's dataset.

Solution

To remove as much contention as possible the goal again is to have a virtual customer store that is physically distributed across a number of disks. Customer information is segmented and partitioned across the available physical disks by a scheme that makes sense to the data set.

In our case customers are segmented into groups based on their area code. The unique reference to a customer is his/her phone number, and the first 3 digits denote the customer's area code, the next 3 digits denote the exchange within the area code that the customer is associated with, and the remaining digits are unique to the customer within the exchange.

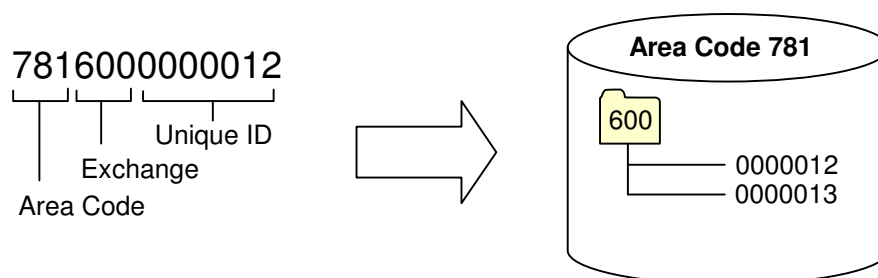


Figure 11 Customer Indexing

As can be seen above in Figure 11 the customer ID is used as the basis for not just what disk this segment of customers are stored to, but also how the customer dataset exists within the data store. That is, the area code is associated with a physical data store, a disk can have multiple physical data stores. The exchange code is used to segment customer data sets within a physical store similar to folders within a file system, each exchange denotes the top node within a hierarchy of customers, and the physical store is made up of many exchanges.

Storing data in this way means that accessing a customer needs no additional indexing mechanism to supplement the physical data. The indexing mechanism is implicit within the storage model, therefore is very quick and also evolves dynamically with no additional index processing necessary (see Strategies below for implementation considerations).

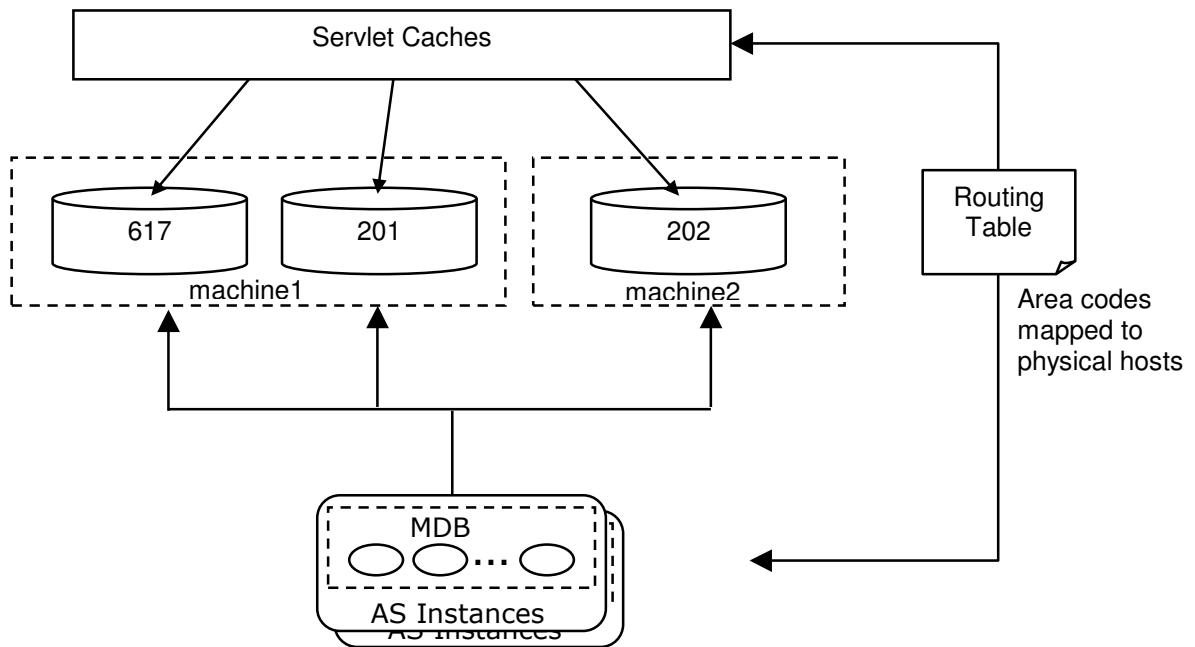


Figure 12 Physical Data Partitioning

The main objective with this pattern is to provide a distribution mechanism for physical data stores across a number of disks/machines, therefore the indexing mechanism has to take this into account. Servlet requests to the customer store first access a routing table that maps area codes (representing physical data stores) to disks attached to specific machines. The servlet requests are then routed to the appropriate machine/store and the customers details retrieved as described above.

Strategies

Our implementation was tested using SonicXIS and ObjectStore. Both of these data management solutions provide a native language storage model, i.e. directly storing Java objects (in the case of ObjectStore) or XML documents (in the case of SonicXIS). This means that we are not confined by a normalized relational structure that needs to be mapped into and out of the application domain. This gave us the freedom to leverage the storage model within the indexing technique used.

This could have been achieved with any data management system, we happened to use SonicXIS and ObjectStore.

Consequences

Retrieving customers from disk is very quick due to the simple and efficient indexing mechanism. Also additional disks can be added across machines as needed at run time without the need to bring the entire system down. Evolving the system can also be done incrementally, evolving one store at a time.

4.5 Message Broker Cluster

The message broker cluster sits between the back end systems and the shared customer information stores. The message brokers role within the system is to manage events generated from the back end ordering system and to rapidly and reliably deliver these events to the customer store.

The message broker cluster could very quickly become a single point of contention as the back end systems grow to accommodate new products and services. Therefore it is crucial that this layer can scale through distribution in its own right.

4.5.1 Broker Clustering and Routing

Context

Events from back end OSS's are propagated to the customer information store. OSS/J API's provide in-built notification mechanism that propagates events across JMS from OSS/J implementations and any interested parties.

Problem

Communicating the OSS/J notification events needs to be scaled along with the back end systems and also as the subscribed applications to the events are scaled.

Solution

The solution is to provide a clustering strategy that allows brokers to distribute load across a number of broker instances while still providing a single virtual interface. In this way groups of brokers forming clusters, may connect to other groups of broker servers as needed, creating highly distributed deployments across loosely-coupled locations. This minimizes the number of servers needed in these configurations to handle large messaging volumes.

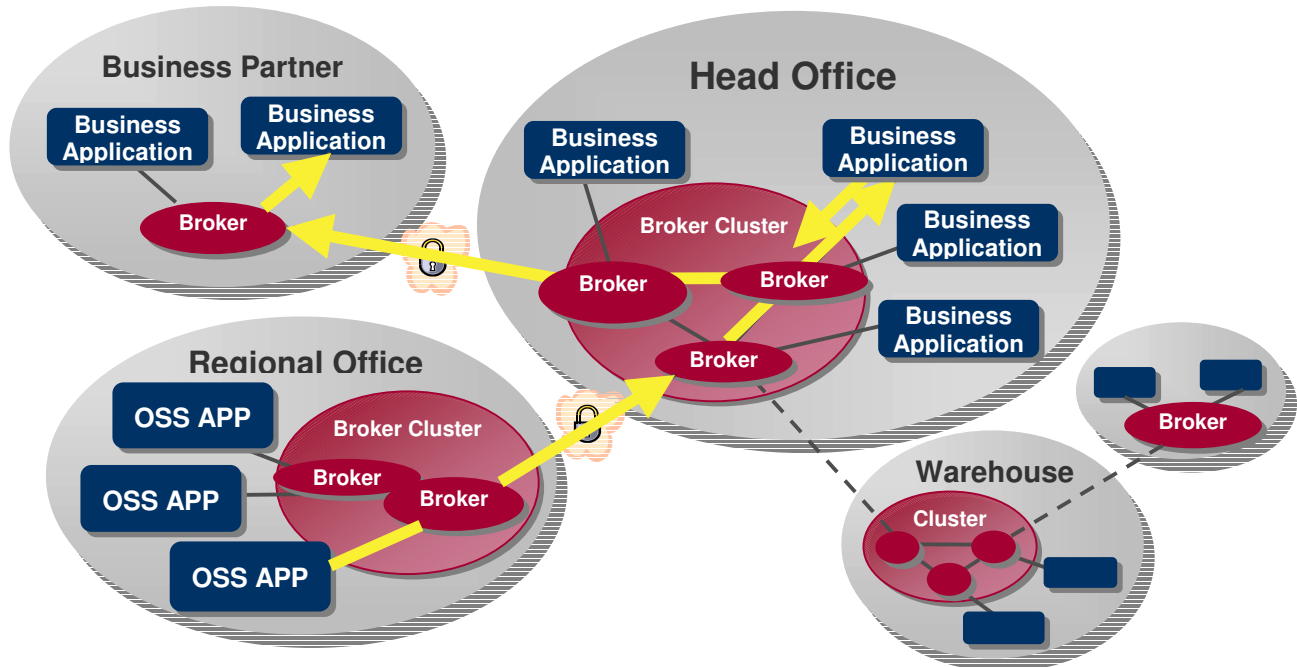


Figure 11 Broker Clusters

Strategies

SonicMQ supports broker clustering through a feature called Dynamic Routing Architecture (DRA). DRA is a messaging server technology that allows increased message volumes to be handled as needed, without reconfiguring application programs or requiring significant administrative overhead. The DRA approach eliminates the need to accommodate topology changes and connectivity issues by dynamically adjusting as needed to support changes in messaging configuration. Additional message servers may be added transparently to support additional external connections, or to scale up internal systems to handle increased message traffic. Cross-cluster communication also provides segregation of destination namespaces and security domains between geographically dispersed locations or application domains while still allowing selective message traffic between the clusters.

Consequences

Brokers can be seamlessly scaled up to accommodate additional load by simple administration.

4.6 Service Activation

The service activation tier represents the back end OSS application that implements the OSS/J Service activation API. This tier manages the customer ordering process, capturing and maintaining the state of orders and mediating with service provisioning systems. Customer order state changes are propagated to the shared customer data cluster above.

4.6.1 Hierarchical Distribution

Context

New orders and existing order state changes must be predictably accommodated, and as the service provider grows its customer base or provides additional services to its customers the service activation application needs to be able to scale with it.

Problem

How do we scale the application logic and order processing to accommodate additional customers or services.

Solution

Split service activation logic into a geographic hierarchy that deals with orders for a subset of the service provider's customer base.

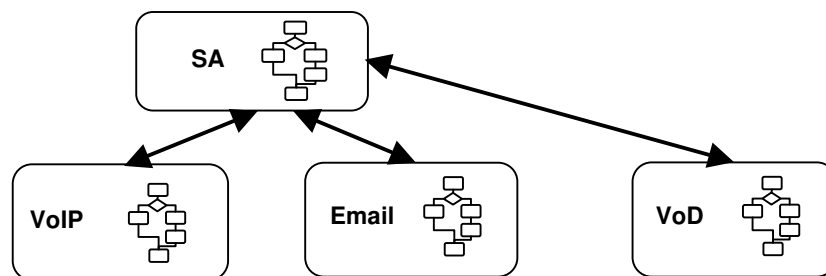


Figure 12 Service Activation Scaling

Also means we can bring on new services dynamically by simply adding a node to the hierarchy.

Strategies

IP VALUE's premioss product with its certified Service Activation API was used throughout the Massive Scalability effort. Premioss has this distribution facility built into the product. So the environment described above can be configured out of the box and evolved as needed.

Consequences

Scalable and future proof migration path to accommodate service provider growth, with clustering support to deal with a higher load, and additional services.

5 Test Methodology

This section details the methodology used to measure the performance and scalability aspects of the Massive Scalability demonstration. We define the interesting characteristics and statistics in the system under test and the constraints under which they must be collected. Also we introduce the tools used to generate the system load and capture the resulting performance metrics.

The methodology consists of the following:

- **Performance Measurement Tools**
- **Performance Metrics Definition**
- **Sampling Method**
- **Running the Tests**

5.1 Performance Measurement Tools - The Grinder

The Grinder is the primary performance measurement tool we used to run the tests for the Massive Scalability effort. The Grinder is a load generation tool and is a pure Java application. The following is a brief description of The Grinder's capabilities.

Three processes make up The Grinder:

- **Agent Processes**, a single agent process runs on each test client machine and is responsible for managing worker processes on that machine
- **Worker Processes**, which are created by Grinder agent processes and are responsible for performing the individual tests
- The **console** which coordinates the other processes and collates statistics

We also introduce two additional concepts:

- **Plug-ins**, Java components that define what a test means and contains the code to execute the tests. For example an HTTP plug-in is provided; so each request performed by the HTTP plug-in corresponds to an HTTP request to an URL.
- The **grinder.properties** file, a text file that is read by an agent, worker processes, and the plug-in, this file is used to specify the number of tests to run and details for each of the individual test – it can be thought of as the “test script”. For example, when using the HTTP plug-in the *grinder.properties* file contains the URL for each test.

The Grinder console is used to control many processes, each running many threads of control, across the test client machines. To run a given set of tests an agent process is started on each test client machine. The agent process is responsible for creating a number of worker processes. Each work process loads a plug-in component that determines the type of tests to run and then starts a number of worker threads. Each of the worker threads uses the plug-in to execute the individual tests that make up the *grinder.properties* file.

The *grinder.properties* configuration file contains all the information necessary to run a particular set of test runs, such as the number of worker processes, the number of worker threads, the plug-in to use, and the details of the tests to run. The agent process and the worker processes read their configuration from *grinder.properties* when they are started.

The net effect of this scheme is to allow the easy configuration of many separate client contexts, each of which will run the same set of tests against our servers. Each context simulates an active user session. The number of contexts is given by the following formula:

(Number of agent processes) X (Number of worker processes) X (Number of worker threads)

Each test invocation is referred to as a *transaction*. A key performance metric in our methodology is throughput which is measured in Transactions per second (TPS). The plug-in defines what a transaction actually is. For example in the context of the HTTP plug-in a transaction is a HTTP request response.

The worker process measure the elapsed time taken for each transaction that is performed (the “response time”). The results for each transaction are recorded in a data file and there is a separate data file for each worker process.

The worker process also records whether the transaction was successful or caused an error to occur. The meaning of “success” is defined by the plug-in. In addition the plug-in can add its own statistics values.

The Grinder console is used to coordinate the actions of the worker processes by sending them start, reset, and stop commands. IP Multicast is used to broadcast the commands simultaneously to processes running on many machines.

The worker processes send statistics reports to the console. The console combines these reports to produce graphs and tables showing test activity. The results of a particular test run can then be saved for later analysis. The console also calculates and displays derived statistics. A key derived statistic that the console can calculate, but the individual worker processes cannot, is a combined Transactions Per Second (TPS) figure for all the worker processes. This is because a rate, such as TPS, can't be calculated without a shared notion of the beginning and the end of the timing period. The console performs the required timekeeping function.

The console provides an easy way of controlling multiple test client machines, displaying test results, and controlling test runs.

More information about the grinder and its use in the context of performance measurement can be found in *J2EE Performance Testing* [8].

5.1.1 Where to obtain the Grinder

The Grinder's homepage is <http://grinder.sourceforge.net>. From here you should click the link for the sourceforge summary page in order to download The Grinder distribution. All of the tests for the scalability effort were run using The Grinder 2.8.4.

5.2 Performance Metrics Definition

There are 2 distinct external interfaces to the demonstration system, and therefore two separate touch points to the system under test to measure:

- **Customer status requests**, spiky and predominantly read only queries relating to order or trouble ticket status
- **Order requests**, to the back end service activation system

5.2.1 Customer Status Requests

In the case of Customer Status Requests our metric must give us a guide with regard to overall performance. This takes the form of analyzing results based on measurements of response time and (AART) and throughput, or transactional rate (in transactions per second, TPS).

AART

The average response time (ART) is calculated for every request in a test script. This is done by calculating the arithmetic mean of every individual response time for every simulated user that runs that test script. The aggregate ART is then the average (the arithmetic mean) of the ARTs of all the requests that make up the test script

Throughput: Transactions per Second (TPS)

This is a measure of the total transactional rate over all the requests in a test script. In this case, a transaction is a request, so we are in fact measuring Requests per Second (RPS): the number of requests processed over a set period of time (in this case one second). It is important to note that this metric is not a “miles per hour” measurement of the speed of the system. It is in fact a measurement of capacity.

Based on an informal study of customer self service, we choose to use **2 seconds** as the maximum acceptable response time for the performance test.

A large tier 1 service provider will have in the order of 10's of million's of customers. Therefore a reasonable user load under production would be in the order of 1 million customer status requests per hour.

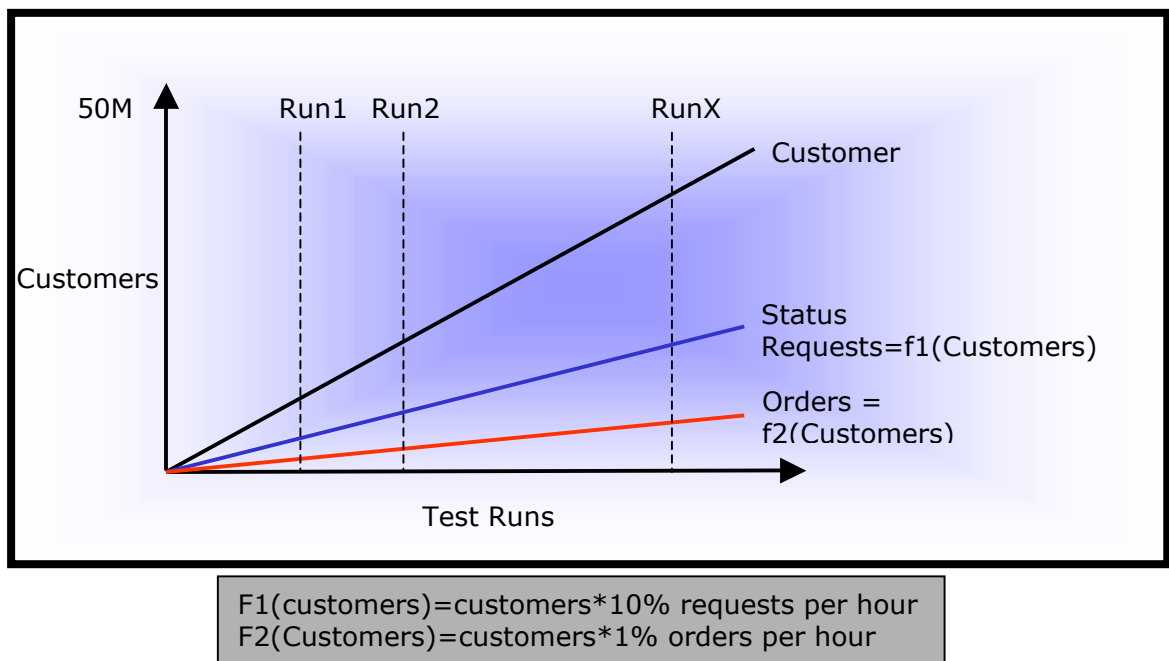


Figure 13 Scaling Strategy

The strategy for scaling the system is quite simple. The number of customers within a service provider will obviously affect the amount of status and order requests. For the sake of the demonstration both status and order requests are expected to be calculated as a function of the number of customers that a service provider currently has. Therefore to scale the system we simply add customers. This maps to what would happen in the real world where, through either acquisition or as a measure of success, the customer base grows.

The Status requests load is defined by the F1 function outlined in Figure 13, basically 10% the current customer figure equals the number of status requests pumped into the system per hour, so for a service provider with 1 million customers the load will be 100,000 customers per hour.

The target is to scale the system to a tier 1 service provider with a customer base of 50 million customers.

5.2.1.1 Customer Generator

In order to simulate the customer interactions we need a way to generate unique customer id's. A simple way to achieve this is to take advantage of a feature of The Grinder HTTP plug-in called **string beans**. String beans allow us to call custom Java methods from a grinder.properties configuration script.

The HTTP requests defined in our scripts call the **getCustomerID()** method from the string bean. The method generates a customerID based on how many grinder processes and threads were running. The getCustomerID method apportions a unique customer range to each worker thread, and each thread then loops over the customer range giving us as broad a customer coverage as possible.

5.2.1.2 Test Scripts

The main aim here is to come up with a technique that can be used for every detailed performance test that models real world customer interactions. The Grinder scripts mimic a customer logging into the customer portal and querying his/her order and trouble ticket status. For the purposes of the demonstration the Grinder scripts contained 10 requests for various customer details, each dynamically gathering the requested data and presenting it back to the customer.

In order to simulate the variable length of time between these requests, depending on the user, we used a 20% variation on the think time between each request within the tests.

5.2.2 Orders Requests

Order state changes from the back end service activation system are propagated asynchronously to a shared customer information store over JMS. Our performance expression is measuring **throughput**, which is typically expressed as transactions per second. In this case a transaction is a message, so we measure the number of messages processed per unit of time, for example messages per second or messages per hour for batch processing.

Unlike the “maximum acceptable response time” metric, which is used for interactive applications based on the perceived quality of the application, throughput is based on an actual business requirement, which can be defined as minimum acceptable throughput, for example an overnight window might exist in which all processing must be successfully completed in order for that company's daily business to resume.

We can measure throughput in two ways:

- The number of message a producer can send to a queue or topic in a specific period of time, typically one second
- The number of message per second the consumer(s) can pick up and process from a queue or topic.

The order load is defined by the F2 function outlined in Figure 13, basically 1% of the current customer figure equals the number of orders pumped into the system per hour, so for a service provider with 1 million customers the load will be 10,000 customers per hour.

It is important to remember that throughput is a measurement of processing capacity, not speed.

5.2.2.1 Test Scripts

The test scripts for order input were quite simple in comparison to the customer portal request scripts. In this case, we simply inject OSS/J create order requests for each customer over JMS.

Each order script generates a single create order request and the scripts cycle over each customer linearly (even though the driver is multi threaded).

6 Results Discussion

This section details the patterns and testing methodology in practice. The following tests were performed at Sun's Garrison labs in Menlo Park, Ca. The tests were split into two separate efforts:

- End to end use case testing, the goal being to scale the entire use case through the variety of distribution and clustering techniques described in section 4.
- A scaled down version of the use case concentrating on tuning a single point within the overall system stack.

6.1 End to End Use Case

6.1.1 Configuration

The hardware configuration used for the cluster tests is depicted in Figure 14 below. The details of the hardware and software used can be found in Appendix A.

The workload was generated using 6 E280R client machines, each running The Grinder worker processes. Each Grinder worker process was using Sun's JDK1.3.1.

A single web server V880 machine running Apache 1.3.7 was the point of distribution for the load generated. The web server used a BEA apache plugin to load balance the user HTTP requests load across the servlet cluster below.

Receiving the load from the web server, the servlet cluster consisted of 3 V880 machines all running WebLogic 6.1 service pack 4. Each machine hosted a single instance of WebLogic which in turn hosted a pool of servlets that made up the customer portal and provided access to the underlying customer shared information. The WebLogic instances clustered and load balanced the HTTP requests between them, in this way scaling to meet the load demands through distribution.

An E280R was used as a WebLogic Admin server to control and administer the other WebLogic instances in the cluster.

Each WebLogic application server instance in the servlet cluster embedded an ObjectStore 6.1/SonicXIS 3.1 service pack 3 customer data cache, providing the servlet pool with a cache of the customer datasets that can be distributed and clustered along with the application server cluster without losing the transactional integrity of the data.

Each servlet cache replicated all customer data, as described in the Transactional Data Caching pattern in section 4.3.2. Further iterations of the Massive Scalability focus group could revisit this pattern and evolve it to the more intelligent *Cache Routing and Data Affiity* pattern described in section 4.3.3.

The physical customer data cluster was running on a single V880 and the customer stores on a mounted T3 with 1 GB of cache. The data cluster initially held 1 million customers, each customer's data set was provisioned with a single order to provide some initial dummy data, which of course grew as the back end order notifications were caught. The customer data was split across 5 physical data stores each containing 200,000 customers.

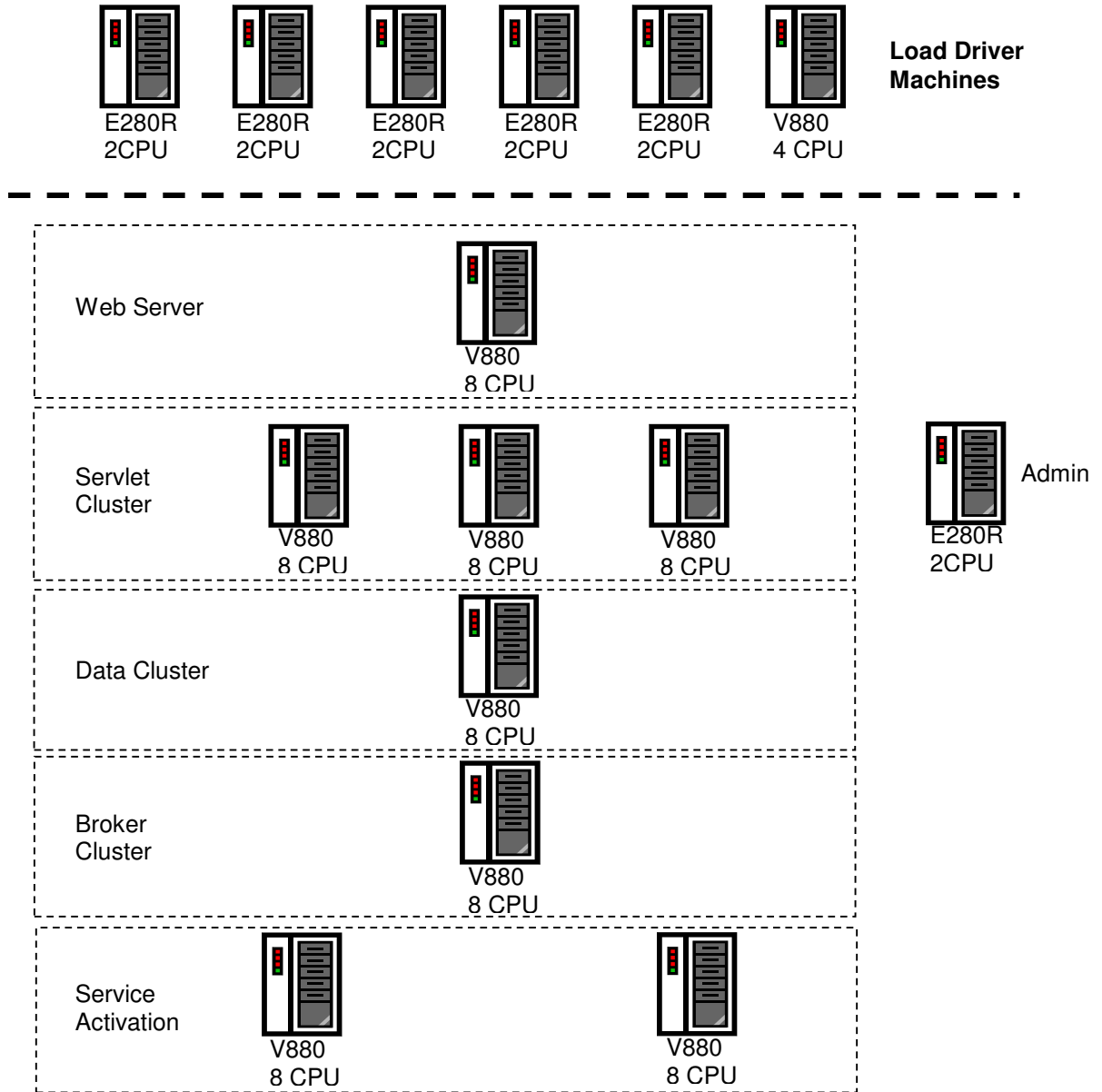


Figure 14 Cluster Case Hardware Configuration

Our broker cluster was implemented using SonicMQ 5 on a single V880. The broker cluster never came under any massive strain in this environment, the majority of the stress was from above the data cluster. SonicMQ was deployed with the default configuration options (no attempt was made to tune it) and was easily able to handle the load with a single broker.

The OSS/J service activation implementation was the premioss service creation product from IP VALUE. Premioss was clustered across 2 V880's.

The back end service activation load was generated using a custom JMS loader that spawned a configurable number of threads that sent an OSS/J CreateOrderEvent to the service activation system and reported the throughput metrics.

Throughout we used JDK1.3.1, and made no attempt at fine tuning specific aspects of the system, e.g. no virtual machine tuning. However the next section will deal with such tuning issues in more detail.

6.1.2 Running the tests

The main scaling strategy for the system relied on clustering the application server to accept more customer HTTP load and also to further protect the back end system. So there were two criteria:

- Scale the system to maintain an acceptable response time to the external customer load
- Do not affect the performance of the back end systems.

The tests were structured into groups of three separate load iterations each iteration increasing the load over the last. The load iterations were scaled by increasing the number of concurrent Grinder worker threads, namely 400, 800 and 1200 concurrent worker threads. Each Grinder worker thread continuously runs the test script outlined in the previous section (see section 5 for more details) for a sequence of unique customers.

We could have associated a thread with a single customer, mimicking a customers behaviour with introduced wait states and then scale the system by adding more threads. But since our anticipated throughput is in the order of thousands of transactions per second, it was impractical to take this approach. So instead each thread represents a sequence of customers (as many as we can throw at the system) with no wait states, effectively trying to stress the system to its limits.

In this way many threads replicates a broad spectrum of concurrent users hitting the system, the more threads we use the greater the concurrent load and also the greater the spread with the customer ranges that are being hit.

Each group of tests was run against a different cluster configuration to assess how effective the distribution patterns are in response to the scaling load. In this case, we mean clustering across all our system tiers, not just the J2EE application server tier, i.e. a combination of servlet and data caching cluster that coexist on the same app server instances. The effect of the scaling load was measured on cluster of 1, 2, and 3 application server instances, where each instance was running solely on a single machine.

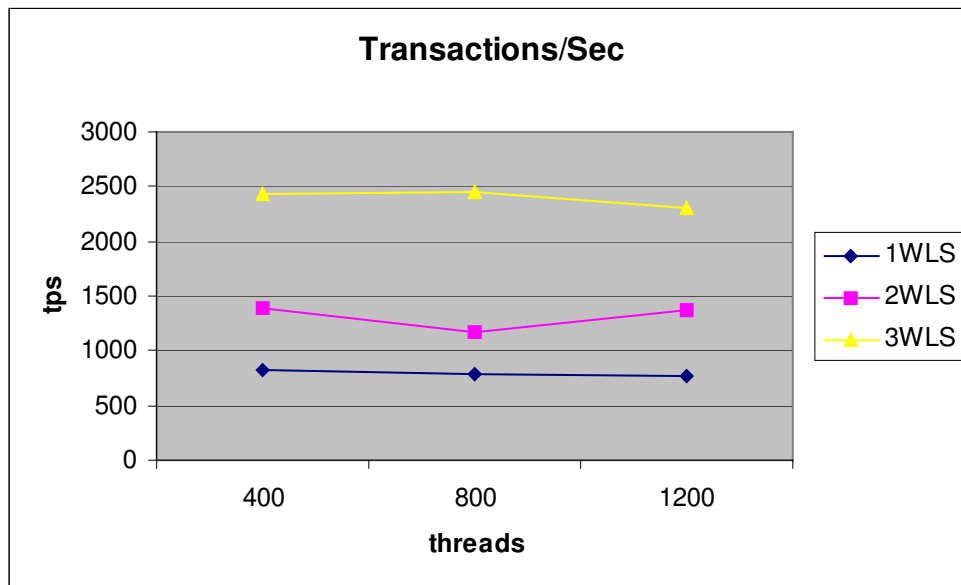


Figure 15 Transactions Per Second

Figure 15 above shows the average HTTP request response transactions per second figures for all tests across every cluster configuration. As you can see a cluster with a single instance can handle approx 800 HTTP transactions per second, and this remains pretty constant as the load increases, that is the degradation in capacity is minimal. With 2 instances in the cluster the transaction capacity increases to approximately 1400 HTTP transactions per second, an increase of 75%, and again the capacity degrades very little. With 3 instances in the cluster the capacity increases to 2400 HTTP concurrent transactions per second, an increase of 71% over 2 instances and an increase of 200% over the single instance case.

NB:

The transaction figures for the cluster with 2 instances are out of step with the other tests, in that the capacity does not scale as much as expected. This was due to an error in the setup of the Apache WebLogic plug-in. The plug-in was setup to load balance HTTP requests across the IP addresses of all machines in the cluster irrespective of which were active or not, and after the tests were run we noticed a number of troughs in the CPU loads of runs where not all machines were active in the cluster. These troughs were caused by the WebLogic plug-in waiting for the inactive machines in the cluster to respond. This anomaly disappears for the cluster of 3 instances since all machines are active. This explains why the cluster of 3 seems to scale better than the cluster of 2.

However the capacity figures are meaningless without the related response times. Figure 16 shows the average response time for all tests.

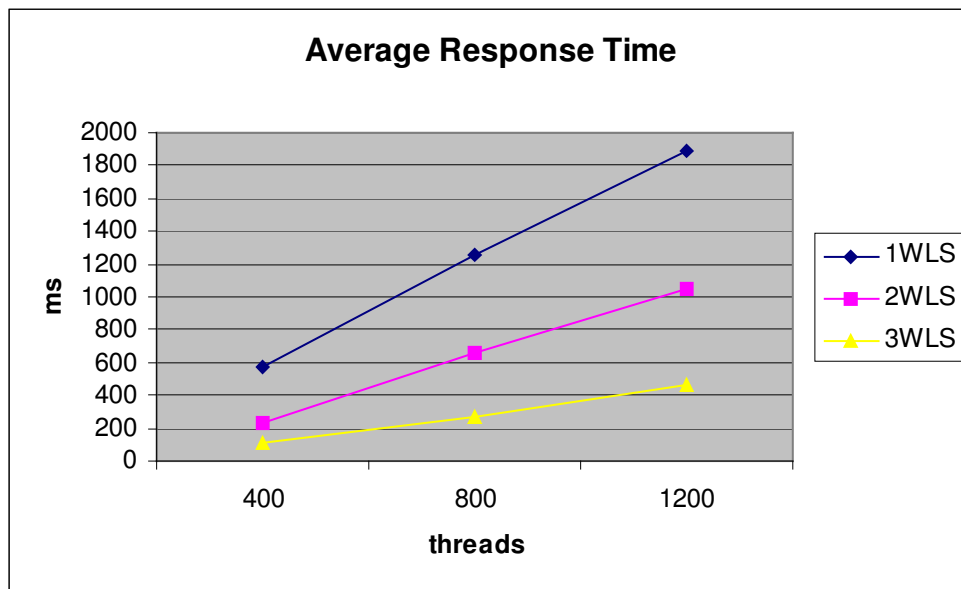


Figure 16 Average Response Time

With a single instance in the cluster we rapidly approach our limit of 2 seconds as the load approaches 1200 concurrent threads. The response time degrades linearly, as can be seen from the graph. Adding an instance to the cluster has the desired effect of reducing the response time, as can be seen the response time decreases proportionally. Adding another instance to the cluster brings the response time down once again and maintains the linear degradation.

Figure 17 shows the order throughput figures for the back end service activation system, which of course is being pounded with order creation requests throughout all of the test iterations. The graph shows the order throughput figures taken over all test iterations. The 1WLS, 2WLS, and

3WLS legends refer to the number of active application server instances in the layers above since this is how the test iterations are driver, rather than relating to the application server instances for the service activation system.

IP VALUE's premioss application was split across 2 V880s, partitioning the work appropriately across each machine and the same order load was placed on the service activation system for each test iteration. As you can see from the graph the service activation capacity numbers (orders per minute) hardly change across any of the load iterations. Reaching throughput numbers up to 4,500 order creation requests per minute. This equates to approximately 270,000 order creation requests per hour.



Figure 17 Order Throughput

The average order execution time, or the amount of time take to process each order creation requests also remained constant throughout the test iterations (as seen in Figure 18 below).

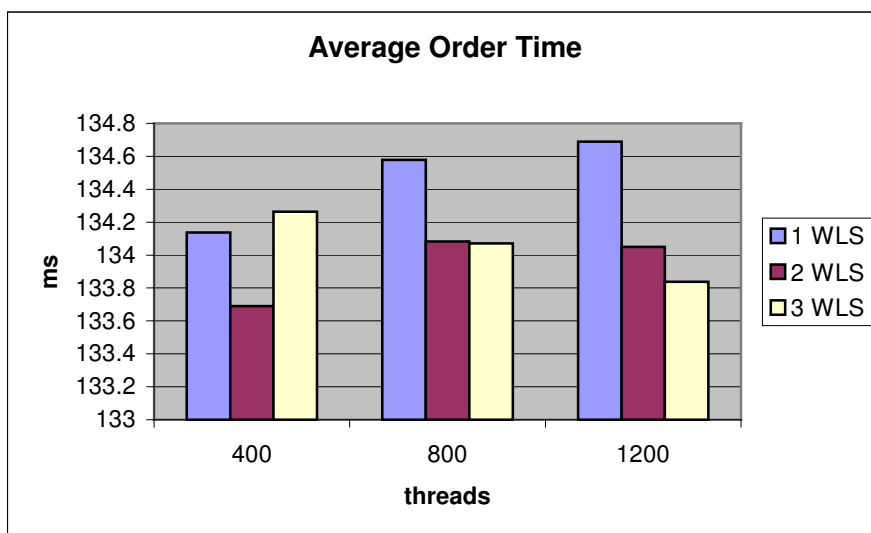


Figure 18 Order Execution Time

The above graphs show very nicely how little effect the increasing load from the customer requests has on the back end service activation system, thanks to the caching and broker clustering techniques employed.

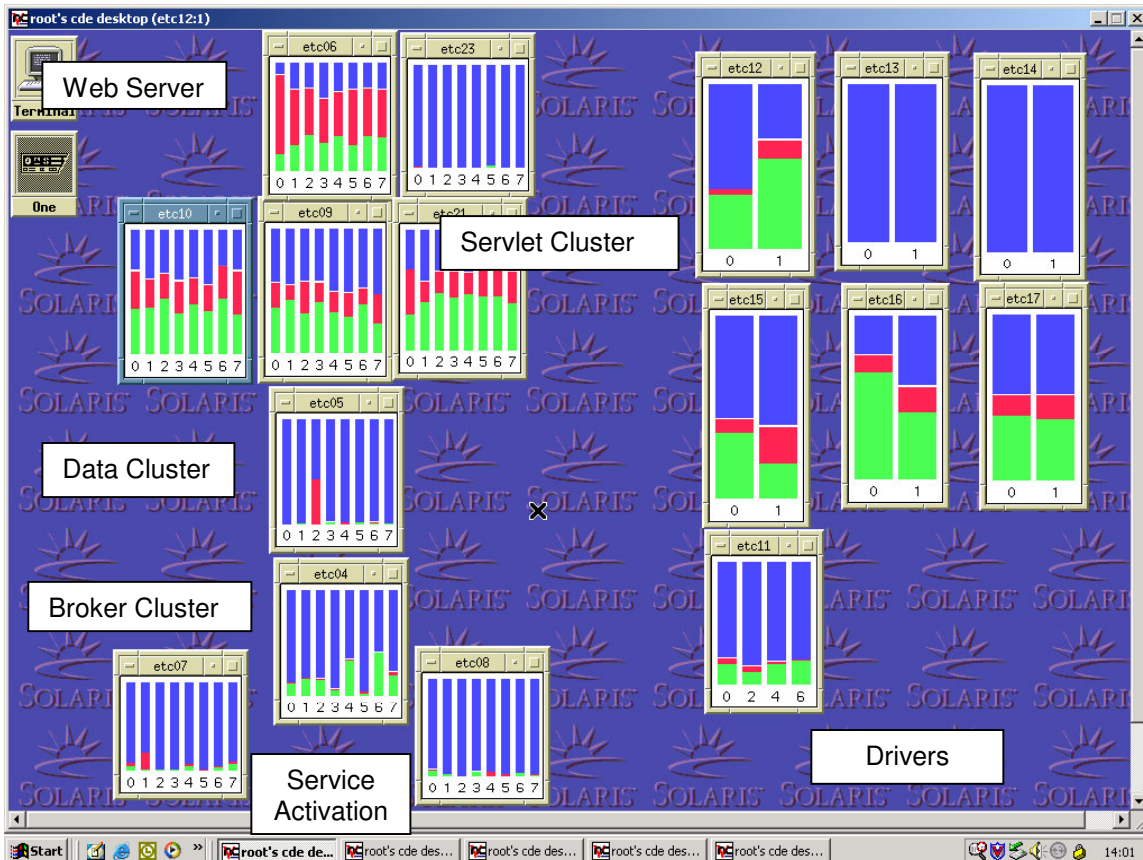


Figure 19 Performance Monitors

Figure 19 shows a screen shot of the CPU performance monitors during a test run. This gives a great view into how the different components within the system are reacting to the load. You can see that the servlet cluster is taking the majority of the load thanks to the caching pattern, leaving the back end ordering system to get on with its duty.

6.2 Single Case

A subset of the hardware that was available for the end-to-end use case was available for further tuning and testing. We took this opportunity to deploy the servlet and data cluster on SunONE 7 application server to get a balanced view of multiple vendor application server implementations. This time also we focused our efforts on system tuning rather than end-to-end pattern tuning as in the previous section.

The cut down scenario did not have the back end service activation functionality that was available in the previous end-to-end tests. The servlet cluster and data cluster were deployed but nothing else, so we were testing on dummy customer information and without an addition web server layer since we had just one application server instance.

6.2.1 Configuration

The hardware configuration for these tests is depicted in Figure 20 below. The load was generated with 2 E280R's and a single V880, again using the Grinder as the load generation and performance/throughput metric capture tool. The servlet and data cluster coexisted on a single V880 and these machines were all deployed on a 100 MB network. Check the appendix for more details on the hardware used.

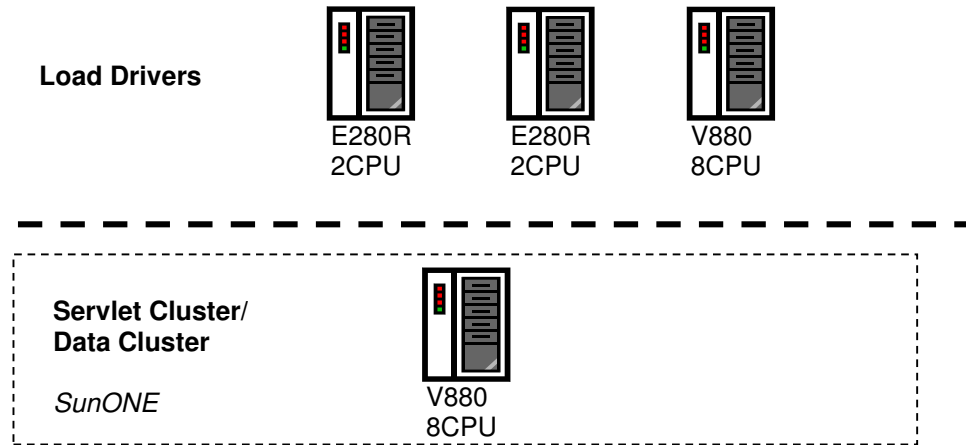


Figure 20 Single App Server Instance

The data cluster was populated with 1 million customers.

6.2.1.1 Java tuning

The default JDK used by SunOne AS7 was changed to JDK1.4.2-beta-b16. Even though this was a beta build, the performance promises warranted an investigation. It was anticipated that applications could see performance improvements of more than 50% to 100% by just using this new release. Applications wishing to take advantage of these performance gains need to use the following garbage collection parameters to see the performance increase.

```
-server -Xms2048m -Xmx2048m -XX:NewSize=256m -XX:MaxNewSize=256m
-XX:MaxNewSize=256m -XX:+UseParNewGC -XX:SurvivorRatio=2000000
-XX:+UseConcMarkSweepGC -XX:MaxTenuringThreshold=0 -XX:PermSize=128m
-Xx:MaxPermSize=256m -Xnoclassgc -XX:+DisableExplicitGC
```

server.xml
<pre><jvm-option> -server -Xms2048m -Xmx2048m -XX:NewSize=256m -XX:MaxNewSize=256m -XX:MaxNewSize=256m -XX:+UseParNewGC -XX:SurvivorRatio=2000000 -XX:+UseConcMarkSweepGC -XX:MaxTenuringThreshold=0 -XX:PermSize=128m -Xx:MaxPermSize=256m -Xnoclassgc -XX:+DisableExplicitGC </jvm-option></pre>

Figure 21 server.xml configuration file

These options were added to server.xml, the config file used by SunOne AS7 (see table above). Details of these options can be obtained by reading the paper, [Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory Using JDK 1.4.1](#)[9]. These options are being combined into an easier option `-Xconcg` and will be

available in the 1.4.2 FCS version. `-Xcongc` will expand to these options and should allow easy tuning of the JDK. The other option that was tried is the `-XX:+AggressiveHeap` option.

The above tunables are all JDK 1.4.1 compatible. The SunOne AS7 performance could have been improved further with JDK 1.4.2 specific tunables, but were not tried during this study.

Notes:

1. SunOne AS7 was run with garbage collection tunables applied, and not the default configuration
2. The Gigabit interface in the lab was not used. Using this could improve the performance as performance seems to hit some bottleneck around 600 threads. Bottleneck was not identified.
3. No orders were created for this configuration. Creating orders should not dramatically affect the read results based on the results of the previous test cases.
4. `Fsflush` was not monitored or tuned.
5. No SunOne AS7 tunables were used.

6.2.2 Running the tests

Similar to the end-to-end configuration the tests were structured into groups of ascending load, starting again with 400 threads, then 800, and finally 1200 threads.

The result of the above configuration changes were quite dramatic in both throughput and response times. Albeit with a cut down version of the end-to-end use case, the only real difference with this servlet and data cluster is the lack of any customer updates. This will definitely have an effect on the performance of the system but based on the results of the end-to-end use case it will not have a dramatic effect. The ratio of customer requests to order updates is very large, and as such the likelihood of a lot of customers hitting an out of date record is quite low. For example in our end-to-end scenario we had approximately 2400 customer requests per second and 80 order updates per second. This system difference however should be taken into account when making any conclusions later.

SunOne AS7 results					
Threads	Transactions Processed	TPS	Mean Response Time (ms)	Peak TPS	Errors
200	541400	1570	77.82	1949	0
400	568657	1860	166.3	2333	0
600	549807	1710	303.3	3975	0
800	529039	1650	438	2150	0
1000	536958	1660	554.8	2303	0
1200	542686	1580	715.7	2042	0

Figure 22 SunONE AS 7 results

Average Response Time for 1 Instance of SunOneAS7 on 1 V880

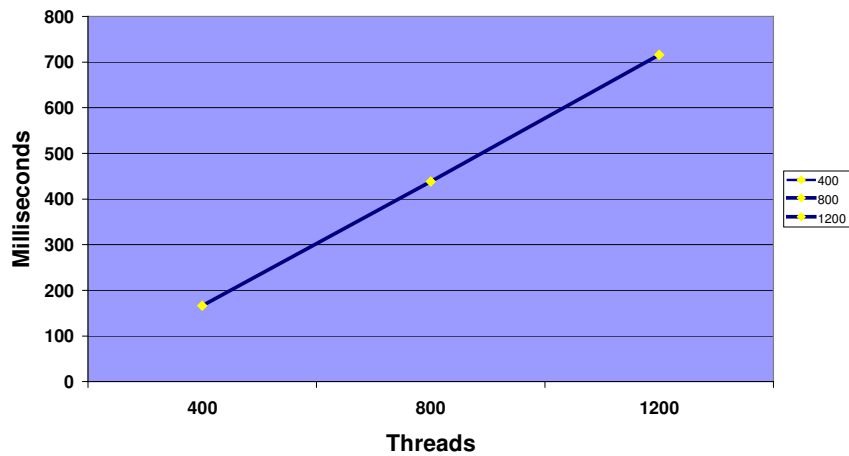


Figure 23 Average Response Time for SunOne AS7

We started out with an initial load of 200 concurrent threads, and as you can see from the table above, the system was able to handle this load quite easily with a request capacity of 1500 Txns / Sec with 80 ms response times. These figures are dramatically greater than the previous scenario, which can really only be as a result of the JVM changes and the scaled down hardware. Presuming that the difference in performance from SunONE AS7 and BEA WebLogic handling requests to the same servlets would be negligible.

Increasing the numbers of threads to 400 the system scaled well, and was able to handle approximately 1860 Txns / Sec with 170 ms response times.

Average TPS for 1 Instance of SunOneAS7 on 1v880

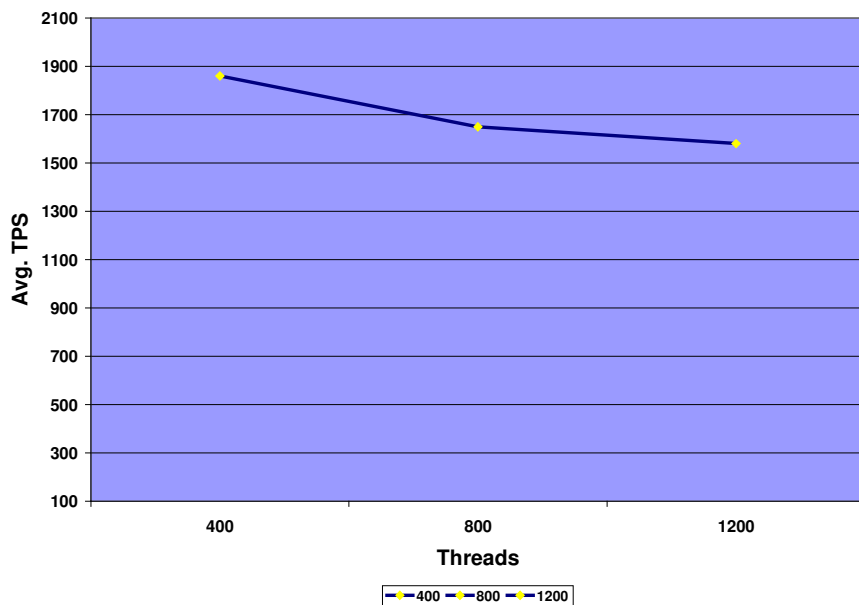


Figure 24 Transactions Per Second for SunOne AS7

At 400 threads the E280R's became the bottleneck with almost 90% CPU utilization. A spare v880, was used for additional load generation and added to the load network. This allowed us to load the system with 1200 threads. The system easily handled this load but the response time had now increased to 700 ms. This figure is well under our allowable limit of 2 seconds.

CPU usage @ 1200 threads

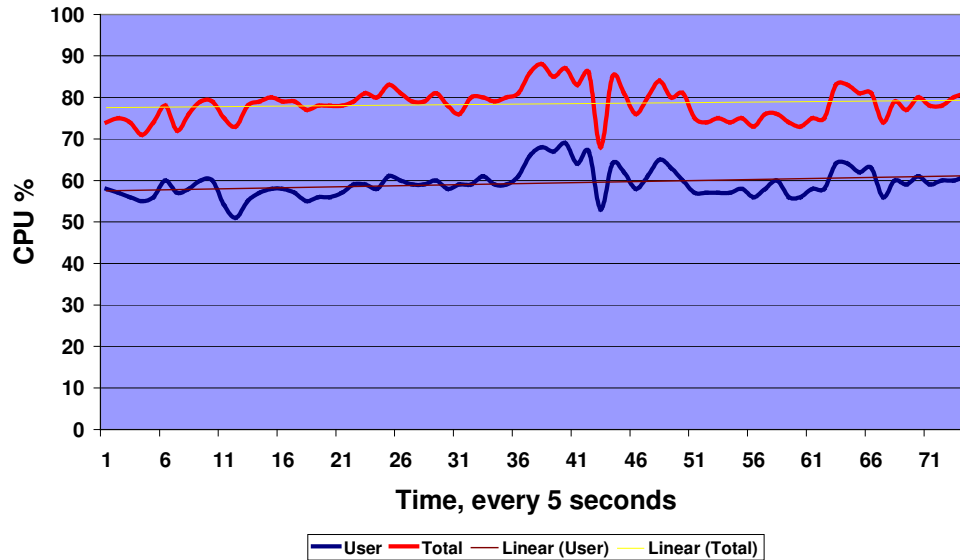


Figure 25 CPU Utilization

Figure 25 above shows the CPU utilization with a load of 1200 threads, the system is petering around 85% utilization.

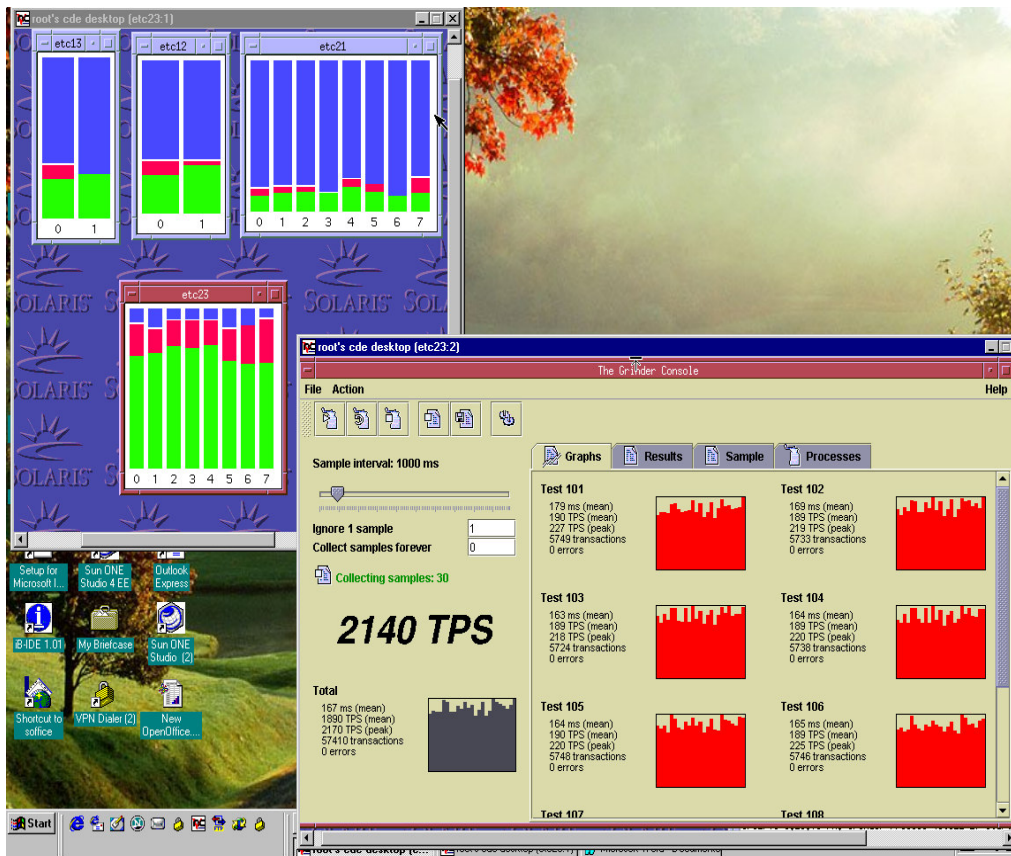


Figure 26 Single Case Grinder Console

Figure 26 shows the Grinder console for the single user case using using JDK1.4.2 Beta (build19) with a load of 400 threads. We took the opportunity to see the effect of the latest available beta of JDK1.4.2 at the time of writing and as you can see from the figure we are able to achieve a constant rate of 2140 transactions per second (an increase of 15% over the previous build 16).

6.3 TCP Tuning

For both system configurations the hardware was tuned specifically to accommodate the dominant use cases, which in this case relied heavily on managing very large numbers of HTTP requests. This in turn of course resulted in very large TCP connection handling duties for the machines.

A S70tcp_tunables script was created in rc2.d directory to set the following tcp tunables at boot.

TCP Tunables
<code>ndd -set /dev/tcp tcp_conn_req_max_q 2048</code>
<code>ndd -set /dev/tcp tcp_conn_req_max_q0 10240</code>
<code>ndd -set /dev/tcp tcp_rexmit_interval 500</code>
<code>ndd -set /dev/tcp tcp_time_wait_interval 1000</code>

The tcp_time_wait_interval tunable was set to 1000 to reduce apache system time. After a few minutes into the test this would increase to 80% or higher. Using netstat -a, and lockstat, we

were able to identify that this was related to the descriptors being in the TIME_WAIT state. The default is 4 minutes.

6.3.1 System tuning

The file descriptor limit was increased by setting the `rlim_fd_cur`, and `rlim_fd_max` variables in `/etc/system`.

<code>/etc/system</code>
<code>set rlim_fd_cur=4096</code>
<code>set rlim_fd_max=8192</code>

7 Conclusion

The end-to-end scenario demonstrated that an OSS/J environment could be scaled quite easily through the distribution patterns employed. Off-the-shelf mainstream IT software products were used throughout to implement the deployment patterns and then scaled through configuration rather than application development. We successfully scaled the end-to-end system across a large number of machines and rapidly changed the deployment configuration to scale up, handling more requests while also reducing the response times.

We showed that through the use of the documented OSS/J deployment patterns it is possible to scale a large system by adding commodity hardware boxes, taking an incremental approach to scaling the system rather than retrofitting the hardware platform with more physical resources such as processors, memory or disks etc.

The system performance and capacity overhead/degradation as a result of employing the distribution patterns was shown to be predictable enough to be suitable for planning purposes. This means of course that a service provider can now plan with a certain degree of accuracy the environment that would be needed to support a certain load (customer requests/ orders or whatever the case may be). Planning operational budgets can now become more of a science than a fine art.

There are of course many improvements that could have made to the system, for example the back end integration, could be better served by taking the distributed JMS to the next level by introducing a JMS-based Enterprise Service Bus (ESB). Also we could have employed the data cluster partitioning and routing pattern to scale the customer dataset to 10's or 100's of millions of customers.

Combining the distribution characteristics of the end-to-end scenario and the fine-tuning results of the single instance scenario, we can make an educated statement that the system would be able to handle a load of close to 4500 transactions per second. The single instance figure topped out at approximately 1800 transactions per second and if we introduce a 15% degradation to be pessimistic about the clustering you get the following formula:

$$1800 \text{ tps} \times 3 \text{ application server instances} \times 0.85 = 4590 \text{ tps}$$

In theory the figures should be even higher than above since clustering the load per server would mean less traffic on each application server instance, which in turn means it should be able to handle more traffic. Also incremental releases of the Java Virtual Machine increased the application server's capacity, in fact as shown in Figure 23 the JDK1.4.2 Beta (build 19) showed a steady transaction rate of 1980-2000 transactions per second. So the new capacity figure would be as follows:

$$2000 \text{ tps} \times 3 \text{ application server instances} \times 0.85 = 5100 \text{ tps}$$

Taking the pessimistic case with a single instance scenario, a steady throughput of approximately 1800 transactions per second equates to 6.5 million concurrent and dynamic (with respect to customer data) requests an hour with a single mid range V880 machine. This is a respectable figure for any tier 1 service provider. The combined figure of 4500 transactions per second with of course a response time of under 1 second equates to 16.2 million customer requests per hour.

Handling 16.2 million requests per hour equates to the entire customer base of a large service provider actively requesting information concurrently within the same hour.

The patterns described within this paper could be applied across all possible OSS/J use cases. They were applied here in the context of a self-service portal, but there is no reason why they would not be applicable to a real-time inventory or billing scenario for example. The next iteration of the Massive Scalability work will concentrate on a completely different OSS use case and add to the library of patterns started in this paper.

Appendix A – Hardware and Software Details

The following is a listing of the computers and software used for the tests in this white paper. The tests were carried out in a clean (the only network traffic was that generated by the tests), high speed Ethernet environment.

Hardware

Type	CPU	Memory	Disk Packs	OS
E220R	2x1024MHz	2GB	Internal	Solaris 8
V880	8x900MHz	32GB	T3	Solaris 8

Software

Software	Vendor
J2EE Application Server	WebLogic 6.1 service pack 4 (http://www.bea.com) SunONE AS 7 (http://www.sun.com)
JMS Broker	SonicMQ 5 (http://www.sonicsoftware.com)
HTTP Server	Apache 1.3.7 (http://www.apache.org)
Java Virtual Machines	Sun JDK 1.3.1 Sun JDK 1.4.1_01 Sun JDK 1.4.2 Beta (http://www.sun.com)
Database/Caching	ObjectStore 6.1 (http://www.objectstore.net) SonicXIS 3.1 service pack 2 (http://www.sonicsoftware.com)
Service Activation	IP VALUE premioss (http://www.ip-value.de)
Trouble Ticketing	TT Reference Implementation 1.0 (http://www.javasoft.com/products/oss)
Load Generation and Monitoring	The Grinder 2.8.4 (http://grinder.sourceforge.net)

References

Design Patterns

1. Sun Java Centre, <http://www.sun.com/service/sunps/jdc/patterns.html>

Benchmarking Methodologies

2. www.tpc.org
3. www.spec.org
4. ECPeef, <http://java.sun.com/>

JSRs

5. OSS through Java™ Service Activation API, JSR89
6. OSS through Java™ Trouble Ticketing API, JSR91

Books

7. Core J2EE Patterns John Crupi, et al
8. J2EE Performance Testing Peter Zadrozny, Philip Aston

White Papers

9. [Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory Using JDK 1.4.1](http://wireless.java.sun.com/midp/articles/garbagecollection2/), Nagendra Nagarajayya, Steve Mayer.
<http://wireless.java.sun.com/midp/articles/garbagecollection2/>

Tools

10. Grinder. <http://grinder.sourceforge.net>

Vendor web sites

11. BEA Systems, <http://www.bea.com>
12. IPValue, <http://www.ip-value.de>
13. ObjectStore, <http://www.objectstore.net>
14. Sonic Software, <http://www.sonicsoftware.com>
15. Sun Microsystems, <http://www.sun.com>