Massey University

J2EE Application for Clustered Servers

--- Focus on balancing workloads among clustered servers

A thesis presented in partial fulfillment of the requirements for the degree of

Master of Information Science

in

Computer Science

at Massey University, Albany,

New Zealand.

Supervised By: **Dr Chris Messom**

Xi Chen

[2006]

Acknowledgment

Special thanks go to my supervisor **Dr. Chris Messom** for his guidance, enthusiasm and technical support during the year. He impressed me with his broad knowledge, background and attitude on every issue. He teaches me not only the latest technologies but also solving problem skills. Thanks a lot.

I also would like to thank the system administrator James Chai for his support to make my project run smoothly, e.g. increasing shared memory size and resetting network routes etc.

Thanks also go to all the developers who have contributed to open source software.

Meanwhile, I appreciate my parents who gave me full support and took care of my sons during my study.

Abstract

J2EE has become a de facto platform for developing enterprise applications not only by its standard based methodology but also by reducing the cost and complexity of developing multi-tier enterprise applications. J2EE based application servers keep business logic separate from the front-end applications (client-side) and back-end database servers. The standardized components and containers simplify J2EE application design. The containers automatically manage the fundamental system level services for its components, which enable the components design to focus on the business requirement and business logic.

This study applies the latest J2EE technologies to configure an online benchmark enterprise application – MGProject. The application focuses on three types of components design including Servlet, entity bean and session bean. Servlets run on the web server Tomcat, EJB components, session beans and entity beans run on the application server JBoss and the database runs on the database server PostgreSQL. This benchmark application is used for testing the performance of clustered JBoss due to various load-balancing policies applied at the EJB level.

This research also focuses on studying the various load-balancing policies effect on the performance of clustered JBoss. As well as the four built-in load-balancing policies i.e. *FirstAvailable, FirstAvailableIdenticalAllProxies, RandomRobin* and *RoundRobin*, the study also extend the JBoss *LoadbalancePolicy* interface to design two dynamic load-balancing policies. They are *dynamic* and *dynamic weight-based* load-balancing policies.

The purpose of dynamic load-balancing policies design is to ensure minimal response time and obtain better performance by dispatching incoming requests to the appropriate server. However, a more accurate policy usually means more communications and calculations, which give an extra burden to a heavily loaded application server that can lead to drops in the performance.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Summary: The chapter describes some brief reasons why the J2EE platform was selected. And then presents the purpose of the study and aims of the study. Finally, the overall structure of the thesis and main contents in each chapter are listed.

1.1 Why Choose J2EE

Since the 1990s, middleware technology i.e. middle-tier software has developed to simplify distributed assembly of components, which is a collection of services including managing communication, security and threads etc. to enable multiple processes on different working machines to interact across the network. For a distributed enterprise application, usually we use middleware to connect separate applications such as linking between web applications to a database system, the technique provides an abstraction capability to simplify the construction of a distributed enterprise system and allows the application developers to only focus on business logic [52]. The most attractive middleware today are CORBA, J2EE and .NET. All of them are deployed as standard components / objects. But CORBA (Common Object Request Broker) provides only middleware techniques to model a standard and consistent component architecture framework such as clients' request for services from servers via well-defined interfaces across network. J2EE and .NET are the two most popular software development platforms to design server-side enterprise applications.

Both J2EE and .NET are emerging and competing platforms that contribute to simplifying writing enterprise applications. .NET is restricted to MS Windows-based platforms although some limited open source multi-platform .NET systems are in development, such as the Mono Project [16]. This research will utilize Massey Beowulf Clustered computers that are running the Linux operating system. In addition, the advantage of J2EE is that it is programmed in Java that can be deployed cross-platform. Moreover, this research extends a previous study, Zhou's Master's research that used SUN ECPerf --- a benchmark J2EE application as the testing application to study "A scalable application server on Beowulf Cluster" [1]. As a result, the J2EE platform is a natural choice for the study.

1.2 The Purpose of the Study

This research addresses techniques that support the development of distributed enterprise applications particularly its non-functional requirements including security, scalability, fault tolerance and load-balancing etc but focuses specifically on load-balancing policies. Due to the dramatically dropping hardware price, using more than one server has become affordable for more and more companies, including most small companies. Moreover, the Internet is providing a potential e-market for enterprises since the network has effectively shortened distances for international or national trade. The quality of e-commerce services, such as securing customer information, ensuring minimal online waiting time to increase returning customers has been a priority. Improving the performance of servers, particularly clustered servers are now being addressed. Load balancing is one of the key technologies behind clustering that affects the performance of the clustered servers when the servers face heavy loads.

The goal of load balancing among clustered servers is to ensure minimal response time and obtain better performance by dispatching incoming requests to the appropriate server. When Zhou [1] studied the scalability of Beowulf clustered servers using JBoss, he found the different load-balancing policies could get different performance results. His study indicated that a better scalability result could be achieved by using First Available Load-Balancing policy than using default Round Robin Load-balancing policy, but both of them became a bottleneck under heavy workloads [1]. In order to improve this situation, this research will focus on studying the load-balancing mechanism. The study will extend the JBoss *LoadbalancePolicy* interface to design a more appropriate load-balancing policy --- dynamic and dynamic weight-based load-balancing policies.

In order to test the performance of clustered servers using various load-balancing policies, this study also includes building a benchmark J2EE application. It consists of:

- Client side: client view of dynamically generated HTML pages using server side technology Servlets that is implemented on Tomcat, a servlet container.
- Server-side: refers to both web server and application server. The web server (Tomcat) focuses on user interface design via web component Servlets and the application server (JBoss) which handles business processes via designing J2EE enterprise Java beans components entity beans and session beans
- Database: used for store business persistent data that is implemented on the open

source database application PostgreSQL

1.3 How the Thesis Is Organized

The thesis focuses on the J2EE application and load-balancing policies design. The entire thesis will be focused on these two main topics and organized into eight chapters. The overall structure is as follows:

- ***Chapter 1*** describes some brief reasons why the J2EE platform was selected. And then presents the purpose of the study and aims of the study. Finally, the overall structure of the thesis and main contents in each chapter are listed.
- ***Chapter 2*** overviews the knowledge background and terminology needed to understand this thesis. J2EE is introduced including the architecture of the J2EE platform, containers and APIs and the structure of deployment of J2EE applications. Secondly the well-known benchmark J2EE application – ECperf design is introduced. And finally the n-tier clustered distributed system and common load-balancing policies in use are presented.
- ***Chapter 3*** covers the hardware system for this study - the clustered systems built in Massey University Albany campus. The detail of the open source software chosen for this study including J2EE application design IDE, web server Tomcat, application server JBoss, database PostgreSQL and performance test tool Jmeter are discussed.
- ***Chapter 4*** details the entire benchmark J2EE application – MGProject design, including database, EJB and web components design. A brief introduction of the EJB and Servlet design, including the life cycle of the EJBs and Servlets, including the main concerns on designing these components is given. In addition, example coding of the design to explain the components are given. Moreover, how the web component and EJB components interact with each other in the application and how to use XDoclet in the NetBeans IDE to generate JBoss specific deployment descriptors is presented.
- ***Chapter 5*** covers the detailed design of the dynamic and dynamic weight-based load-balancing policies for clustered JBoss. The policies will extend JBoss *LoadBalancePolicy* interface and utilizes EJB timer service and JBoss T*reeCache*. The *LoadBalancePolicy* interface exposes the cluster members' information. The EJB timer service performs tasks in a regular period and JBoss

TreeCache provides shared storage for all timer session beans located in different machines. In addition, the chapter also lists the pseudo code design.

- ***Chapter 6*** The test plan in this chapter has two meanings. One refers to establishing a test plan in Jmeter. The plan will simulate a large number of independent clients performing online actions to interact with back-end servers. Another means to plan a test procedure for testing the performance of JBoss under various load-balancing policies. This plan should provide a sequence of testing steps to achieve the testing goal. Meanwhile, techniques to ensure the benchmark application runs as smoothly as possible under the very heavy loads are discussed.
- ***Chapter 7*** presents the details of the tests implementation on the Sisters cluster. The main tests are classified into two kinds – scalability tests and load-balancing policy comparison tests. The scalability refers to a single JBoss and clustering two or more JBoss performance test. The load-balancing policy tests will apply various load-balancing policies on the EJB level to identify if each of them affects the final performance of clustering JBoss.
- ***Chapter 8*** gives the final conclusions of the thesis. The conclusions include a critique of the J2EE application design and the effect of the various load-balancing policies on the system performance. Finally, the chapter also presents potential further work.

Chapter 2: Background Overview

Summary: The chapter overviews the knowledge background and terminology needed to understand this thesis. J2EE is introduced including the architecture of the J2EE platform, containers and APIs and the structure of deployment of J2EE applications. Secondly the well-known benchmark J2EE application – ECperf design is introduced. And finally the n-tier clustered distributed system and common load-balancing policies in use are presented.

2.1 J2EE

J2EE stands for ***Java 2 Enterprise Edition***. It was introduced by Sun Microsystems, Inc. in 1998, which attempted to extend the design concept of Java language "Write Once, Run Anywhere" to "Write Once, Deploy Anywhere". The concept is presented in "*understanding Java and the J2EE platform*" [2]. It developed from the J2SE (Java 2 Standard Edition) platform where developers could develop stand-alone applications with Java and its built-in libraries but did not have a standard method. Only a few years later, J2EE has become a de facto platform for developing enterprise applications. It is not only a standard but also has reduced the cost and complexity of developing multi-tier enterprise applications. Moreover, J2EE is able to deliver highly available, secure, reliable and scalable applications that meet today's enterprises requirements.

2.1.1 J2EE Platform

J2EE is a broad and deep subject. As a platform, it contributes to designing server-side distributed applications. It defines a standard way to combine APIs and deployment tools. It also combines some programming languages, protocols and techniques in an enterprise server project under the J2EE specification, such as HTML, XML, JDBC, SOAP, RMI/IIOP, JSP and build tool ANT etc. On the other hand, the platform aims to simplify the development of business applications. It uses containers to manage lower-level resources and communications, which let the developers focus on business logic.

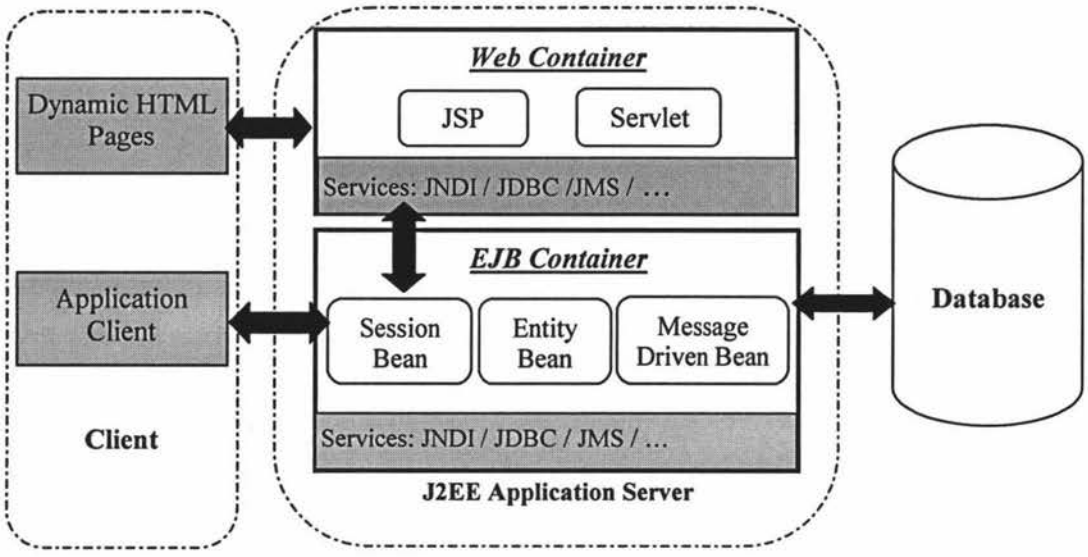


Figure 1: Overview of J2EE Application Architecture

Furthermore, the platform provides a foundation for designing multi-tier applications; it develops a set of specific reusable components that are different from the normal Java classes. They are JavaServer Pages (JSP), Servlets and Enterprise Java Beans (EJB). Each kind of component is good at handling various business logics and gets the services from its container. In short, the J2EE platform provides a collection of standardized components, containers and services for creating and deploying distributed enterprise applications within a well-defined distributed computing architecture [3].

Figure 1 shows the overview architecture of J2EE application. From the architecture, it is obvious that [3, 4, 6]:

- J2EE is a multi-tiered architecture. The application server keeps business logic separate from the front-end applications (client-side) and back-end database servers.
- All components exist in a container that is responsible for providing the runtime environment and services to its components, such as JNDI (Java Naming and Directory Interface). There are two kinds of containers in an application server, the web container and the EJB container.
- The web container manages web components: Java servlets and JSP. Both generate dynamic HTML pages. The main difference between them is that usually servlets handle more dynamic data than JSP, i.e. servlet is Java code with some static HTML and JSP is HTML with some Java code. In the J2EE application server, the JSP is compiled into a servlet first and then is executed by

the web container.

- The EJB container hosts enterprise Java bean components: session beans, entity beans and message driven beans. The main functions for these beans are: Session beans model business processes such as business logic and rules. Entity beans represent persistent business data that is stored in one row of database table. Message driven beans are designed to be able to asynchronously handle the processing of incoming JMS (Java Message Service) messages.
- The bi-direction arrows (in figure 1) indicate the best way of communication among J2EE application tiers and components. In fact, J2EE application server offers more flexible communication, e.g. all components can directly interact with the database.

Details of the J2EE platform are given in "*the J2EE tutorial*"[2].

2.1.2 The J2EE Containers and APIs

The most significant aspect of J2EE is that the use of components and containers. The detail of the components design is presented in Chapter 4. The container itself is an interface between a component and the low-level platform-specific functionality that supports the components [3]. That means, in order to simplify the development of J2EE applications, the container separate business logic design from system-level issues such as multithreading handling and resource management and let the developers concentrate on business logic rather than the system infrastructure. Corresponding to J2EE application component types, there are four types of containers, i.e. application client containers, applet containers, web containers and EJB containers [3]. The former two kinds of containers are located in the client machine and the latter two kinds of containers are as in figure 1 located in the application server. The latter containers are the focus in this study.

The containers need to provide some services to its components. So it has to access a set of enterprise services via the J2EE standard APIs to find an instance, process transactions and access database pooling connection etc. So the J2EE standard APIs actually specifies a contract between the containers and J2EE applications [6].

The implementation of J2EE is standardized under the J2EE specification (the latest version is 1.4). All J2EE vendors, including commerical platforms like BEA WebLogic and IBM Websphere or open source J2EE platforms like JBoss and JOnAS. All vendors can implement the vendor-specific APIs to access various enterprise services. But all of

them at least provide the following APIs and support the services under the specification J2EE V1.4 [7]:

- JDBC (Java DataBase Connectivity) API that allows access to various databases from the components except from applets.
- Java IDL (Interface Definition Language) API allows the applications that were written in any languages to access any CORBA (Common Object Request Broker Architecture) object via standard IIOP protocol, which provides standards based connectivity and interoperability.
- RMI-IIOP (Remote Method Invocation - Internet Inter-ORB Protocol), which allows objects defined using RMI style interfaces to be accessed using IIOP protocol.
- JNDI (Java Naming and directory Interface) API performs lookup for J2EE objects in a specific application, which providing a unified interface to multiple naming and directory services in an application.
- JAXP 1.2 (Java API for XML Processing) that enables the applications to validate XML documents against a particular XML parser.
- JAAS (Java Authentication and Authorization Service) is a set of APIs that enforce access controls and authentication, which support user-based authorization.

The above APIs has been part of J2SE platform that the containers have to support. J2EE V1.4 also requires the specified APIs version that the vendors also need to implement. They are *EJB 2.1, Servlet 2.4, JSP 2.0, JMS 1.1, JTA 1.0, JavaMail 1.3, JAF 1.0, JAXP 1.2, Connector 1.5, Web Services 1.1, JAX_RPC 1.1, SAAJ 1.2, JAXR 1.0, J2EE Management 1.0, JMX 1.2, J2EE Deployment 1.1 and JACC 1.0.* But it is not required that all containers have to implement the above APIs. For example, none of them is required for the applet container, and Servlet 2.4 and JSP 2.0 are only required for the web container [7]. The specification of EJB 2.1 and Servlet 2.4 specify a framework development of the EJB and web components design that is presented in the MGProject benchmark application.

2.1.2.1 EJB 2.1

The EJB specification specifies the enterprise Java beans component-based framework for development and deployment of multi-tier distributed applications. Since

EJB 2.0 was released, the EJB design concept has been significantly changed. These changes focus on four aspects: integration with JMS, using CMP / CMR (Container Managed Persistence / Container Managed Relationships), introducing local interfaces and offering inter-server interoperability [9]. Using CMP/CMR and local interfaces are the most important concepts in EJB component design [56].

1. *Using CMP / CMR*: the development of applications has been simplified by creating an abstract schema to isolate the application from the physical database schema. That means that the container can fully complete the methods *get(), set(), findByPrimaryKey()* and handle relationship etc from and to the database and keeps the data consistence. That means the developers don't need to code reading, writing data and the relationships betweens beans or recoding them when the underlying data structure changed.
2. *Introducing local interfaces*: the local interfaces mechanism improves performance of the enterprise beans that are located in the same JVM. The enterprise beans in the same JVM can call each other directly instead of using RMI calling. But the calling semantics of local interfaces are different from the remote interfaces. Since remote interfaces use call-by-value semantics, while local interfaces use call-by-reference. Moreover, local interfaces provide the foundation for CMR among entity beans since the EJB specification specify that to use CMR, the local interfaces must be exposed [20].

The EJB timer service is a new feature in the EJB 2.1 specification. The EJB timer service is an event-based mechanism for scheduling business event to invoke EJBs at a specific time, such as generating reports, reading data or doing audit work. This service can be used with entity, stateless session and message-driven beans but not for stateful session beans [14]. A bean invokes the timer service via the enterprise bean container, i.e.

```
TimerService timerService = context.getTimerService();
```

A timer can be created or actived via this TimerService interface. For entity beans, the timer is associated with the bean's identity, which is a particular instance of the bean. That means if an entity bean sets a timer in *ejbCreate(...)*, then each bean instance will have its own timer. In contrast, stateless session and message-driven beans do not have unique timer for each instance.

The timer can expire once or many times. In the *javax.ejb* package there are four interfaces related to timers [14, 48]:

- TimedObject: defined only one abstract callback method *ejbTimeout()* to deliver timer-expiration notifications. So the EJB bean implementation class has to implement this interface.
- Timer: it contains information about a timer created via the EJB Timer Service.
- TimerHandle: it handles a Serializable object used for persisting Timer information.
- TimerService: it exposes the EJBContext interface to allow EJB components to be able to access the timer service that is managed by the EJB container.

A bean invokes *createTimer()* methods via timerService object, e.g.

```
timerService = context.getTimerService();
Timer timer = timerService.createTimer(intervalDuration, "created timer");
```

Timers are persistent and saved in the default database of the J2EE application server. The EJB container will automatically call *ejbTimeout* if the J2EE server is restarted for any reasons [48]. The EJB timer service used in this study is a stateless session bean to refresh system load average values for the clustered application servers for the dynamic load-balancing policies.

2.1.2.2 Servlet 2.4

The traditional method used to get dynamic web contents was through the Common Gateway Interface (CGI), which failed to deliver scalable Internet applications since CGI has to create a new process for each client request [5]. In 1996, Sun Microsystems introduced Servlet technology that enables Java code with some static HTML to be loaded dynamically into and run by the web server to provide object-oriented abstractions for building dynamic web applications. The dynamic contents are based on the client's requests and the response incorporate data from databases or other resources. The Servlet interacts with clients via a request-response model based on HTTP protocol. Compared to traditional web applications, the users have to wait for the entire web page to be reloaded from the server. Dynamic web pages only need to refresh dynamic content without the entire page being reloaded. This should benefit on saving waiting time, saving bandwidth. Finally the benefit will save the money for business. The comparison study and the study showed a business could save between 500 and 2800

man-hours per year [24].

The Servlet 2.4 specification provides a framework to develop a Servlet. It specifies the Servlet interfaces, life cycle and deployment descriptor and so on. The Servlet 2.4 was slightly upgraded from the existing version 2.3 and added some new features, such as it depends on HTTP/1.1 and J2SE 1.3 and is able to work with J2EE 1.4, the deployment descriptor web.xml file uses XML schema and so on [17].

Servlet filters and filter chains are a powerful technique that was introduced in Servlet 2.3. This technique is used to design common parts for all the web pages like header, navigator and footer in the benchmark application. A Servlet filter and filter chain is [18, 19]:

- A filter is not a servlet, so it doesn't actually create a response.
- A filter can dynamically intercept requests and responses to transform or use the information contained in them, such as modifying the request / response headers.
- A filter pre-processes the requests before it reaches a servlet and post-processes the response after leaving a servlet.
- A filter can be attached to one servlet or a group of servlets and one servlet can also have zero or more filters.
- A filter chain is a mechanism for invoking a series of filters by the container.

2.1.3 J2EE Deployment Structure

To deploy a complete J2EE application in an application server, the application and components need to be packaged into modules. The J2EE specification provides guidelines for the structuring and the creation of J2EE applications. The application module *.ear* file is composed of any number of sub-modules, i.e. EJB module *.jar* file, web module *.war* file, application client module *.jar* file and resource adapter *.rar* file [4]. All modules at least contain a standard XML file called the deployment descriptor, which contains information about the deployment structure of the components. These standard deployment descriptors are located at a specific directory. Figure 2 shows the module view of the benchmark J2EE application and its deployment descriptors. Moreover, the J2EE vendors also specify their own deployment descriptor. For example, for JBoss EJB module, it has *jboss.xml* and *jbosscmp-jdbc.xml*, and for its web module, it specifies *jboss-web.xml*.

The module structure is given below [4, 6]:

1. EJB module (*.jar* file): is packaged into a Java Archive file that consists of all the deployed EJB and deployment descriptors. This module contains business logic that will be deployed into the EJB container. The standard deployment descriptor for this module is *ejb-jar.xml* that is located in the *META-INF* directory.

2. Web module (*.war* file): is packaged into a Web Archive file that consists of a Servlet, JSP, images and some resource files and a deployment descriptor. This module provides the front end to allow users to interact with the business EJB components that will be deployed into a web container. The standard deployment descriptor for this module is web.xml that is located in the WEB-INF directory.

3. Application module (*.ear* file): contains the complete enterprise application that consists of the collection of *.jar*, *.war* files and resources files and a deployment descriptor. This module will be deployed into an application server, for this study it is deployed into */deploy* or */farm* (for clustered servers) directory of JBoss. The standard deployment descriptor for this module is *application.xml* that is located in the *META-INF* directory.

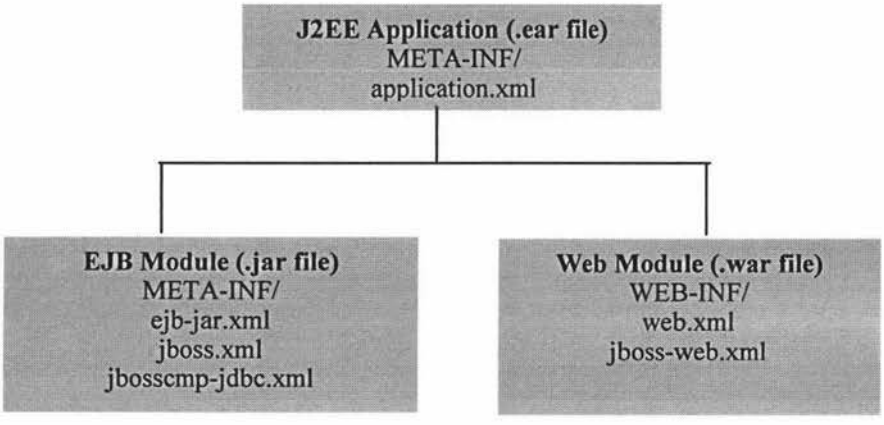


Figure 2: Modules view of a J2EE Application

2.2 Benchmark J2EE application

2.2.1 ECperf

Due to the increasing complexity of J2EE application servers and evaluating the quality of a vendor's product, it becomes necessary to measure the quality of an application server, such as its performance, transactions handling, database connectivity etc, although the different vendors almost all claim to have highest performance and most scalable product under their own test suite. Sun's ECperf benchmark test kit is regarded as the most suitable test kit to measure the performance of J2EE application servers. Deshpande and Martin (2001) provided 8 reasons why ECperf is the right choice to evaluate the performance of J2EE servers [21]. And Zhang, Liu & Qu (2003) concluded ECperf model was more complex than the real world use case and activities and more similar to enterprise wide IT system after comparing IBM's Trade2 with Sun's ECperf [22].

The Standard Performance Evaluation Corporation (SPEC) released three versions of J2EE benchmark application that each cost $2000. But SPECjAppserver 2001 and SPECjAppServer2002 were basically a repackaging of the ECperf. The new release version SPECjAppServer2004 has been enhanced by modifying the workload and adding more J2EE 1.3 standard capabilities. The significant change in SPECjAppServer2004 is accessing the application via the web layer (in dealer domain) and EJB (in the manufacturing domain) replace the old versions that directly communicate to the EJB [23].

Fortunately, ECperf is open source software and SPEC also publishes SPECjAppServer2004 design document, which provides a guideline to design a J2EE benchmark application that will be used to test load-balancing policies.

2.2.2 The ECperf Design

ECperf was written by Sun Microsystems under the Java Community Process program that measures the scalability and performance of J2EE servers and EJB containers. The application consists of a set of EJB, JSP, HTML documents, data schema and drivers etc. The beans were programmed using two versions CMP and BMP that adhere to EJB 1.1 specification. The latest release version of ECperf is v1.1.

ECperf models a realistic business system, a large global enterprise in four domains, i.e. corporate, customer, manufacturing and supplier domain. Each domain has a

separate database and application. The workloads in the four domains (see figure 3) could be configured into a centralized or a distributed model [54]. In the centralized workload model, all domains' database is combined into a single database. And each domain would manage its own database in the distributed model. The bold arrows in figure 3 indicate the relationship in these four domains. For example, when a customer makes an order, this activity occurs in the customer domain, the corresponding information has to be passed to the corporate domain to check if the customer has sufficient credit, what kind of discount he/she can get etc. The information also needs to pass to the manufactory domain to schedule the order in the work plan, to classify the order if it belongs to a large order, and update the order status before assembling and after completing the order. Meanwhile, the manufacture domain would be modified according to the order to purchase required material from the supplier domain.

The heart of ECperf is the System Under Test (SUT), which consists of the application server (only EJB container), database servers and network connections etc. The drivers (refers to codes that drives the benchmark and records all relevant statistics and reports) and supplier emulator (consists of servlets that can be deployed on any web server) reside outside the SUT [54]. The communication to the SUT must be accomplished over the Internet using any standard protocol supported by the EJB container, e.g. RMI/IIOP etc.

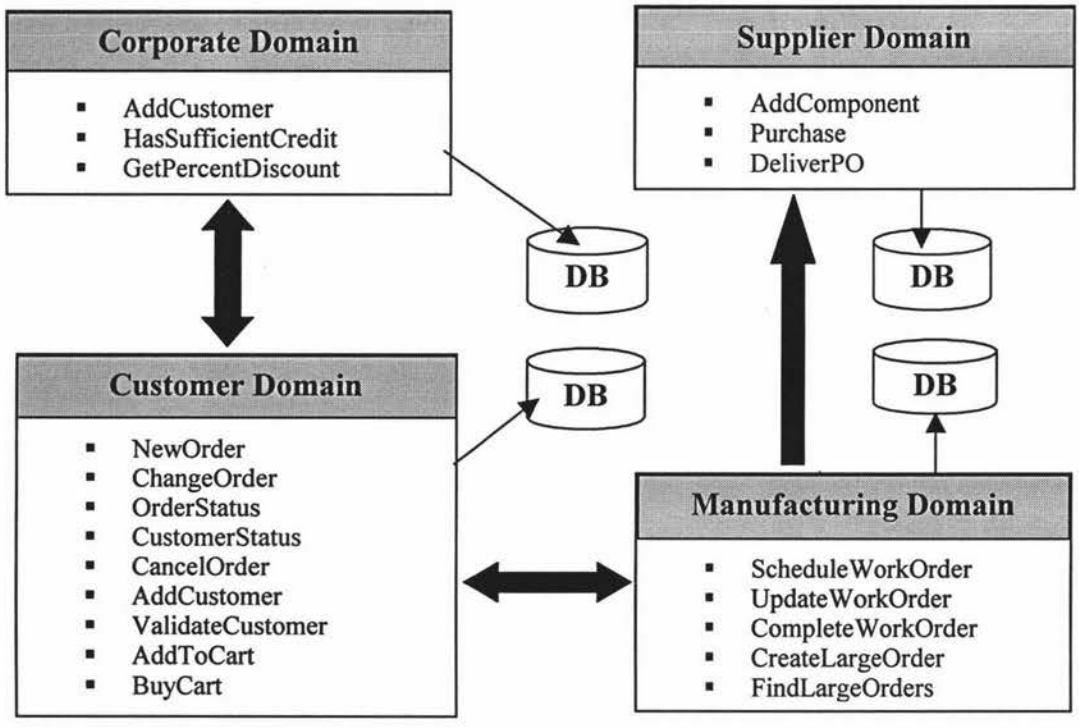


Figure 3: Four Domains Workloads in ECperf

2.3 Load-balancing Policies in Distributed System

2.3.1 Distributed System

Nowadays, large-scale computing environments require many computers to improve the performance and solve the tasks that only one would not be able to handle. A distributed system is the best solution, which refers to those physically independent computers specifically configured together via a network and distributed middleware to appear as a single coherent system to its users (see figure 4). The World Wide Web is an example of a distributed system. The topologies of distributed system can be centralized, decentralized, rings, hierarchies or hybrids configuration [26]. Such a system can be fault tolerant and more powerful than many combinations of stand-alone computers.

The main goals of a distributed system includes [25, 27]:

- ***Transparency***: the system appears to users as a single computing system. The system should have access transparency, location transparency, resource migration and relocation transparency, concurrency transparency, failure transparency and keep data persistence.
- ***Openness:*** the system offers services according to standard rules, e.g. the passing message has a standard format, syntax and semantics etc. Such a system has capability to be extended and interact with other system.
- ***Scalability:*** the system can easily be altered to accommodate a different number of users and have resources located at far geographic areas and are easy to use and manage.

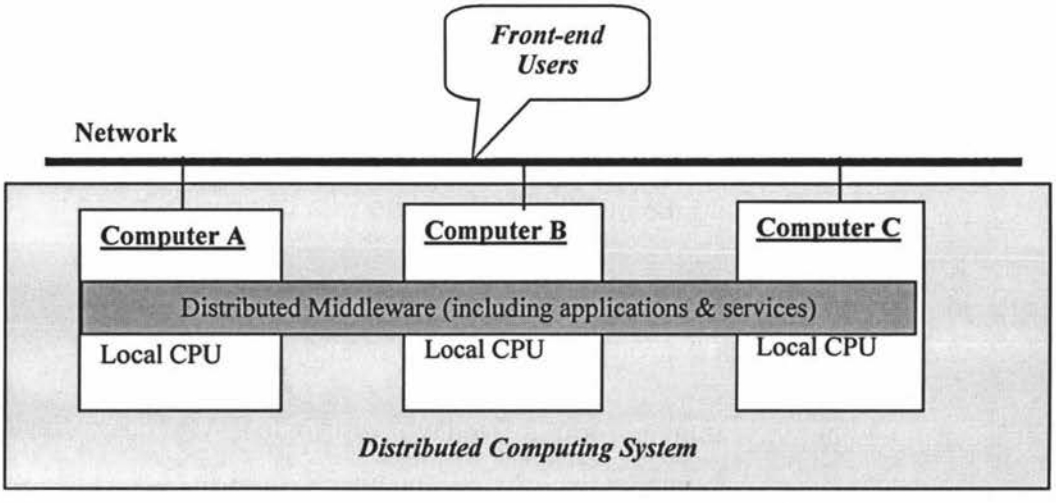


Figure 4: Middleware View of Distributed System

The architecture of the distributed system can be implemented on various hardware and software configurations. The well-known forms of distributed system have client-server, n-tier architecture, clustered and peer-to-peer etc. This study focuses on clustered J2EE application server systems, which is a clustered n-tier architecture distributed system.

2.3.1.1 Distributed System with N-tier Architecture

The earliest form of distributed system was the mainframe with its dumb terminal. This is a one-tier distributed system. Later systems develop from fat / thin two-tier client-server, three-tier distributed system to today's n-tier computing system. The concepts for building the business application are high coherent modules and loose coupling between modules.

In a three-tier system, the front-end client implements the presentation layer. The business logic stays at the middle-tier that runs on the application server. And the back-end data resides on the database server. But in n-tier systems, the layer classification depends on the complexity of the business logic. The architecture usually breaks down into a user interface, presentation logic layer, business logic layer, infrastructure services layer and the data layer [6]. The advantage of developing n-tier (n>=3) applications over the traditional applications includes [6, 55]:

- Easy to modify and maintain, i.e. when new requirement are added, we only need to build new components without rewriting the entire application.
- Multiple user interfaces can be built and deployed without changing the application logic codes.
- Database connection can be "pooled" and reused, which reduce the connection time and improve efficiency.
- The application can be distributed since the communication among different layers is via standard protocol. E.g. the client interacting with middle-tiers may use HTTP / RPC, and the middle-tier interacts with database server via standard SQL/JDBC/ODBC.

2.3.1.2 Clustered Distributed System

Clustering is another popular form for today's distributed system. A cluster is a group of redundant computers that work together via software and network

configuration as a single system to achieve the common goals including [25, 26, 27]:

- ***High availability*** refers to the high possibility of each request can get immediate response.
- ***Scalability*** sees sections 2.3.1 for an explanation.
- ***Load balancing*** means to dispatch incoming requests to different machines so all machines have equal workloads.
- ***Fault tolerance*** implies high availability. If a machine cannot work, other machines can do the same job and give the same correct response. But fault tolerance has stricter requirements on data correction than high availability.

Usually, in a clustered system, to cooperate having the same state within the clustered members is very important. Typically, the clustered members communicate with each other either by point-to-point TCP/IP using RMI or sending state via multicast IP address [32]. Most J2EE application servers implement IP multicast e.g. JBoss and WebLogic. The implementation of session state replication has two common methods. One is that session state is replicated only if the session state has been altered in a server, the server will broadcast the new state to clustered members. Another way to replicate the state is in a fixed timer, when the timer ticks on, the state of session will be replicated automatically.

All major operating systems support clustering, such as Microsoft includes its clustering application directly into Windows 2000 advanced server OS. Most Linux cluster systems develop from the Beowulf cluster and have similar configuration. Other well-known clusters types are COW (Cluster of Workstation) and Mosix [31]. This study uses the Massey University IIMS (Institute of Information, Mathematic and Science) Beowulf cluster computers – Sisters.

The main features of Beowulf cluster system [29, 30]:

- The system usually consists of one server node with several client nodes. And the client nodes don't have keyboards or monitors in most cases.
- Usually it is a group of personal computers running on open source UNIX-like operating system, such as Linux and BSD (Berkeley Software Distribution).
- It is a parallel computing system. The programs are usually written in C and FORTRAN. The communication commonly is via MPI (Massage Passing Interface) and PVM (Parallel Virtual Machine) to achieve parallel computation.

- The cluster is configured into a small TCP/IP LAN with Ethernet interconnection.

2.3.2 Common Load-balancing Policies

The load balancing is a mechanism that will allocate equal amount of tasks to all available computers in a cluster system. For example, for web cluster servers, load balancing implies that there are many clients making requests to target servers concurrently. A load balancer sits between the clients and servers and makes decisions to dispatch the requests to which server. In general, all clients can get served faster because of higher availability and higher performance should be achieved with an appropriate load balancer. A load balancer could be implemented with hardware, software or a combination of both.

2.3.2.1 Hardware Load Balancer

The hardware load balancer shows high available via rewriting requests' header and adding dispatched machine's IP address. But for outside connections, the load balancer shows a single virtual IP address to the clients [33]. The illustration is shown in figure 5.

So all clients' requests will come to the hardware load balancer and then get redirect (amending the request header) to one of server nodes in the cluster based on the load balance algorithms in the balancer. The advantage of using the hardware load balancer is that if one machine like serverA goes down from the cluster, the client request will only hit serverB or serverC by the load balancer. The IP address remains the same since all cluster members appear with the same IP address to the clients.

However the disadvantage of a hardware load balancer is that it is complex to configurate and the device is expensive. One of the important requirements of a cluster is that it should be able to handle session state consistently. When serverA goes down, all state information will be lost. So even if the subsequence requests are dispatched to other available servers, some information may be incorrect. In addition, the single hardware load balancer will be a single point of failure, i.e. if the hardware load balancer goes down, the entire system will fail.

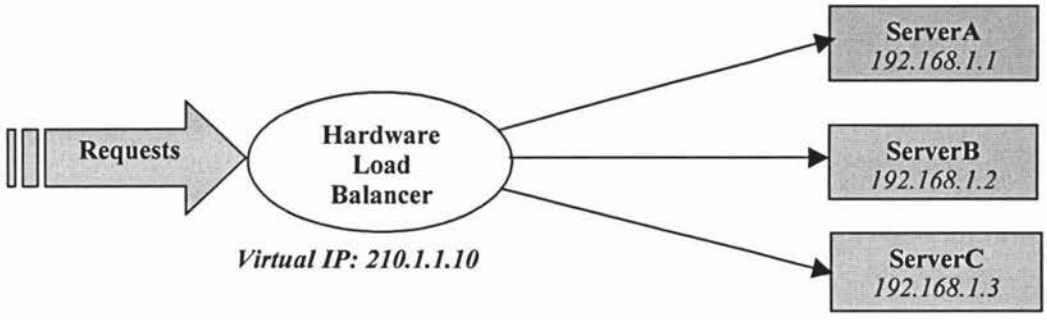


Figure 5: Working Principle of Hardware Load Balancer

2.3.2.2 Software Load Balancer

Due to the complexity and cost of the hardware load balancer, many load-balancing services are achieved in software. Usually, the software load balancer is implemented in different ways for different kinds of servers or applications. For this study, two kinds of servers – web server Tomcat and application server JBoss have been examined. The clustered web server Tomcat and the clustered application server JBoss use different mechanisms to control load balancing among clustered members. When the load balancing mechanism is implemented in clustered Tomcat via the HTTP server Apache, Apache does the job likes a hardware load balancer. But the JBoss clustering realized via farming (hot deployment), each JBoss instance has the same replicated information about the clustered family when it starts and pre-selected load-balancing policy that will be downloaded by a client before the client sends a request to a server. So the clients implement the load balance mechanism in a JBoss clustering environment. For detail about implementing load-balancing mechanism in both servers see chapter 3.

Whatever the hardware or software load balancer, the load balancer implementation always depends on the load balancing algorithms selected, i.e. load-balancing policies. Currently, the common load-balancing policies include [32, 33]:

- ***Round robin policy:*** it is a simple, cheap and very predictable policy that is based on the round robin algorithm that selects an available server node from a list of clustered servers in order. The first server is selected by random chance. After selecting, the server will move to the end of the list and the next server will be selected for the next client.
- ***Random policy:*** each server node is selected by random chance. In such a case, one of the servers may be assigned more jobs than others. But when the cluster

faces heavy loads, the probability of selection for all nodes tend to evenly balance. The policy is recommended only for homogeneous cluster deployment.

- ***First available policy:*** always allocates the first available server node (the first server selection is selected by random) to serve a client. So the client will stick on the same server if the server is always available. But when the server node is shutdown or dead, the new randomly selected node becomes the next first available node.
- ***Weight-based load-balancing policy***: it is more advanced policy compared to the above policies. It takes into account a pre-assigned weight for each server. The weight value will determine what proportion of the load each server should take. The disadvantage of the policy is that it is computation complex and needs to carefully consider the relative weight value.
- ***Minimum load:*** always allocate the clients' tasks to a server node with minimum workload. But the idlest server node may become the heaviest nodes in a short time. So such load-balancing policy should be implemented via frequent comparison of load values in the clustering environment.

Although the J2EE specification specifies the containers and components design, it does not specify the load-balancing policies in a J2EE clustered application servers' environment. Different vendors implement their own selection policies. For instance, the WebLogic implements three load-balancing policies, round robin, and random and weight-based policies. And JBoss prefers to use round robin, random robin and first available policies.

Chapter 3: Hardware & Software for the Study

Summary: the chapter covers the hardware system for this study - the clustered systems built in Massey University Albany campus. The detail of the open source software chosen for this study including J2EE application design IDE, web server Tomcat, application server JBoss, database PostgreSQL and performance test tool Jmeter are discussed.

3.1 Hardware

An appropriated cluster environment is an essential requirement for this study. At Massey University Albany campus, there are three powerful parallels computers – *Sisters, Helix* and *Double Helix*. The department of Institute of Information and Mathematical Sciences (IIMS) is responsible for maintaining the Sisters and Helix. Both of them are distributed memory parallel computers i.e. Beowulf Clusters.

The Helix was ranking in the top 500 supercomputers in the world when it was built. It was configured with Athlon MD MP-2100 processors that linked one main server and 65 client nodes via a fast Ethernet Switch. The main server Helix0 has a 2GB memory and five 36GB hard disk drives. Each client node has a 1GB memory and a 40GB hard disk drive [40]. The facility was built and maintained for research throughout Massey University, so at most time, all nodes in Helix are very busy.

Therefore, although Helix and double Helix are more powerful clustered system with more nodes, they are not suitable for doing this research since Helix and double Helix are busy most of the time. The Sisters is a smaller Beowulf Cluster that has a similar architecture to Helix. The system runs the Linux operating system and communications via MPI protocol. This small clustered system was built for IIMS staff and post-graduated students research.

The current Sisters is configured with a main server node "sisters" but with only 8 client nodes. All machines in the cluster run the Linux operating system. All have dual CPU processors but with slightly different capability (see table 1). The network connection within them uses a Netgear Gigabit Ethenet switch [40].

	sisters	amd1, amd2	amd3, amd4, amd7, amd8	amd5, amd6
Model	PIII	AMD 1900+	AMD 2100+	AMD 2200+
RAM	1GB	1GB	1GB	1GB
(one) CPU (MHz)	845.842	1600.38	1733.745	1800.427

Table 1: Hardware in Beowulf Cluster - Sisters

From table 1, the server node 'sisters' has the lowest configuration, amd1 - amd8 represents each independent client node in the Sisters. All client nodes have higher CPU processing capability than the server node.

For a distributed system, the communication overhead with low bandwidth is the critical issue that causes the performance of the entire system to reduce. The fast Ethernet ensures high-speed links among the clustered members to reduce the network traffic delay. All the cluster systems in Massey University use Gigabit network adapters.

3.2 Software

All software the study used is open source software. As the name suggestion, the basic idea behind the open source software is very simple, i.e. the software source codes can be read, modified and redistributed. All interested developers can improve and debug the software. The quality of open source software is usually very good.

Moreover, the attractive advantage of the open source software is that it is cost free. And the categories of open source software almost cover every field giving a wide range choice. Of course, lack of documentation and lack of direct technical support are the shortcomings of the open source software. But this disadvantage has diminished since the wide spread use of the Internet and often we can research problems and get the solution from the web. Therefore, for this research, selecting open source software is a wise choice.

The selected open source software have been developed several years and have a reliable history and good documentation. The following sections introduce them in turn.

3.2.1 IDE – NetBeans 4.1

NetBeans is an excellent Integrated Development Environment (IDE) to design Java applications, which is developed by Sun Microsystems. It is written in 100% Java and can run on various operating systems. NetBeans use IDE generated Ant build scripts to build, test and debug the various applications, such as general Java applications and Web applications etc. When the 4.1 version was released, NetBean has plug-in J2EE

modules for the Sun Application Server, which makes writing J2EE applications easier [35].

All Java programs need a suitable version Java Virtual Machine (JVM) to run. NetBeans 4.1 requires Java Standard Development Kit (JDK) 1.4 or above. For the installation of the IDE see *Appendix A*. The latest IDE version 5 beta or higher can directly integrate with the JBoss 4.0 application server, see Using NetBeans with the JBoss Getting Started Guide [34].

NetBeans 4.1 needs to explicitly use XDoclet, which is a source code generation engine via special JavaDoc tags to integrate other application severs like JBoss. XDoclet can parse source code and its JavaDoc tags to generate XML descriptors, which make it easier to code J2EE applications since all J2EE applications need XML-based deployment descriptors. The XDoclet tags include many widely recognized J2EE patterns and vendors, example J2EE tags include @ejb, @web and vendor specific tags @jboss, @websphere, @weblogic etc.

When creating a new enterprise application in NetBeans, many jobs can be done automatically. It directly creates the EJB-Module and Web-Module; under each module it allows creation of any number of folders; it can automatically generate all standard deployment descriptors i.e. application.xml, ejb-jar.xml and web.xml; it can connect to a database and generate container manager persistent entity beans directly from database tables; if we define the relationship in tables with foreign keys, the NetBeans can keep the same kind of relationships in the entity beans and produce the standard deployment descriptors.

3.2.2 Database – PostgreSQL 7.4

PostgreSQL is an object-relational database system with standard SQL, including SQL92 and SQL99. It has two versions - open source free version and commercial products version. This study only concerns the free version. It has a powerful and reliable architecture and is one of the world's most advanced open source databases since it provides a wealth of features that are usually only found in commercial databases such as Oracle and DB2 [13]. It requires an Unix-like operating system, until version 8.0 that supports Windows.

Basically the PostgreSQL only require a small amount of disk space less than 150 MB installation room, which grows with the amount of data stored in the database. But PostgreSQL is sensitive to the size of the shared memories buffer since it uses shared

memory for carrying out its work. If the size of the shared memory is too small, postgreSQL will use the disk so degraded the performance. More shared memory does not necessarily give better performance since it also has a threshold that is determined by other system processes and activity. So "as recommendation, shared memory should be no more than 25~50% of system RAM unless the system is a dedicated database server, in which case you can try up to 75% of system RAM " [15].

The default maximum number of concurrent connections to the PostgreSQL 7.4 is 100. This is an important parameter setting but it is kept as low as possible. Since increasing this parameter may cause PostgreSQL to request more system resource, such as system shared memory or semaphores than the operating system's default configuration allows [12]. If a web application has persistent connections, the connection pooling should be used to reduce the number of database connections required and make the connection reusable.

PostgreSQL configures a postmaster to handle communication between the SQL client and the server. Connections can be made to the database server PostgreSQL via JDBC driver *org.postgresql.Driver,* which provides a standard Java API to access the services provided by the PostgreSQL. In most cases, the JDBC connection requires TCP/IP socket to be enabled i.e. the server starts with option "-i". The connection information is kept in the *pg_hba.conf* file under the *data* directory.

3.2.3 Web Server – Tomcat 5.5

Apache Tomcat is developed and released under the Apache Software License. It was recognized as one of the best application servers in 2003 by InfoWorld readers [36], which implements the Java Servlet and JSP technologies via a Servlet container, i.e. a web engine. So some open source J2EE application servers integrate and bundle the web server with Tomcat, such as Sun, JBoss and JOnAS.

Tomcat versions upgrade to match the release of Servlet and JSP specification. The latest version Tomcat 5.5 implements the specification of Servlet 2.4 and JSP 2.0. The installation and configuration of standalone Tomcat is simple (see Appendix D), particular for web archive files *.war*. In such cases, the archive *.war* file is dropped into the Servlet container --- *<tomcatHome>/webapps/.* Tomcat will automatically expand *.war* file into a folder. In order to execute the application, addition library files are required. In tomcat, it provides two locations for these [46].

- *<tomcatHome>/common/lib*: this lib is visible for both web application and

internal tomcat.

- *<tomcatHome>/shared/lib*: this lib is only visible for web application. So it is the right place to put the library files that are only used for the web application. This application, *jbossha.jar, jbossallclient.jar*, *jbossj2ee.jar* and *MGProject-EJBModule.jar* are needed by the web application.

3.2.3.1 Tomcat Clustering

Configuration of Tomcat clustering is more tedious (see Appendix D) since it needs an Apache HTTP Server and a load balancing mechanism. Tomcat 5 or above provides three different mechanisms to handle load balancing: using the JK native connector, using Apache 2 with *mod_proxy* and *mod_rewrite* or using the balancer web application [32]. The JK connector is in front of the clustering Tomcat. The benefit of utilizing the JK connector is that any of the Tomcat servers can be implemented independently i.e. stopped and started without affecting users. The mod_jk / mod_jk2 connector mechanism is used to realize the load balancing in Tomcat clustering.

The concept for establishing Tomcat clustering is shown in figure 6. The communication among Tomcat clustered members use multicast ping message [32]. That is all Tomcat instances will broadcast its IP address and TCP listening port number in the message when making session replication. The clustered member will be considered shutdown if other members do not receive messages from the member within a given time. Since Tomcat 5.0 was released, the cluster feature is built in with all-to-all session replication form, which means the session attributes are propagated to all cluster members. The bold bi-directional arrow in figure 6 indicates the communication processing among the clustered members to replicate session state. Meanwhile, when we implement the clustering, the load-balancing policy is taken into consideration. The connector *mod_jk2* acts as a dispatcher to connect with Apache HTTP server and clustered Tomcats, which transparently fails over to the users.

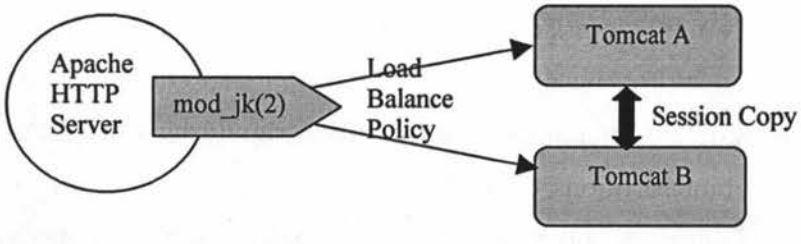


Figure 6: Tomcat Clustering

3.2.4 J2EE Application Server – JBoss 4.0

JBoss is a popular J2EE application server. The recent survey from the 4th annual Java use and Awareness study, JBoss AS is regarded as the preferred Java platform for the enterprise ahead of IBM WebSphere, BEA WebLogic and Oracle. It is the only application server whose market share is on the rise and above 30% [10]. Making JBoss an attractive choice is not only that JBoss is standards compliant, cross-platform and free to download, but also it combines industry-leading technical support with advanced tools such as JBoss networks to manage and monitor enterprise applications.

JBoss claims that it is the first open source application server officially certified J2EE 1.4 and obeys the formal J2EE specification. The web server Tomcat 5.5 is bundled within JBoss AS 4.0. The latest version of JBoss is V4.0.3 that can be implemented with the latest EJB 3.0 but requires JDK 5.0 to install and run. So JBoss always presents itself as the trusted leader in open source middleware software [41].

The application server is the most important part in a J2EE application. So in order to run the benchmark J2EE application successfully, it is necessary to examine JBoss in detail. From the standalone JBoss installation to configuring clustered JBoss, and to control concurrent access to a third-party database system PostgreSQL etc, all aspects need to be analyzed. *'Getting started with JBoss 4.0'* [42] provides a quick tour of JBoss, from the server structure, configuration to its services, which provides a basic knowledge about single JBoss. *'The JBoss 4 application server guide'* [43] provides more details. Appendix C gives details about successfully installing and configuring a single JBoss v4.0.3 and clustering JBoss. The following sections present the main features of JBoss that are relevant to this project. These features mainly include implementation of JBoss clustering, load-balancing policies, transaction handling and tree cache.

3.2.4.1 Clustering in JBoss

Clustering JBoss adds redundant JBoss servers to the cluster group, which provides a mechanism to scale the enterprise application in order to achieve fault tolerance and improved performance. JBoss uses the term *"partition"* to describe a group of JBoss instances in the same cluster family. Each partition should have a unique name since JBoss allows different partitions on the same network. A possible but more complex case of a JBoss instance is that the instance may belong to more than one partition at the same time. So in most cases, each JBoss instance always is allocated to a single partition. The 'DefaultPartition' name will be used as partition name if it is not specified

in JBoss clustering, i.e. if all JBoss instances are started without a partition name, then all instances would belong to the same default partition. The current JBoss provided clustering functionality includes [43, 44]:

- *Auto-discovery of each other without extra configuration:* JBoss uses JavaGroups to realize the dynamic node discovery when a node joins or leaves the partition.
- *Fail-over and various load-balancing policies apply for stateless session beans, stateful session beans, entity beans and JNDI:* for detailed implementation see chapter 4.
- *State replication for stateful session beans:* JBoss uses in-memory replication to synchronize the state of stateful session beans across cluster nodes.
- *HTTP session state replication for web application server:* the web server has to keep same state information across the cluster.
- *JNDI tree replication:* HA-JNDI tree will be replicated across the cluster that is the entry point to look up the home interface of EJB.
- *Dynamic JNDI discovery:* EJB home interface looks up always through HA-JNDI and then delegates to local JNDI.
- *Hot deployment via farming for .ear, .war and .jar files*

Farming is one of main attractive features of JBoss clustering, which provide cluster-wide hot deployment - automatically duplicated across all nodes in the cluster family. The hot deployment is implemented with two aspects, one is that the deployed application will be pulled locally by all member nodes in the same cluster at start up time or when a new member joins in. When the deployed application is deleted from the */farm* directory in one of the running cluster server, then the application will be undeployed locally and then be removed from the cluster family. In other words, farming will take or remove the deployed files across the entire cluster that ensures that all clustered members have the same version *.ear*, or *.war* or *.jar* file.

3.2.4.2 Load-balancing Policy in JBoss

J2EE was introduced to simplify developing large-scale enterprise applications. Clustered J2EE application servers were introduced to achieve mission-critical services, i.e. fault tolerance and better performance. Such mission critical services will rely on the load-balancing policies selection, since load-balancing policy selection and implementation become the critical issue in a cluster environment. Which node or

component is selected to serve requests at run time will be based on the load-balancing policy. Such a policy can be used at the web container level, at EJB container level and at the database level. In JBoss, the web container level load balance implementation is via the bundled web server Tomcat connector mod_jk. And database level load balancing is usually implemented via the database itself. In this case, all cluster JBoss will access the same database PostgreSQL. There is no load-balancing policy needed in such a case. As a result, this study focuses on EJB level load-balancing policies implemented in JBoss application servers.

The J2EE application server JBoss utilizes a smart client proxy to manage fail-over and load balancing logic implementation. JBoss always requires the client to download stub codes that contain a list of available nodes in the cluster family and pre-selected EJB level load-balancing policy via HA-RMI [43, 44] when the client requests services from a JBoss. The mechanism is illustrated in figure 7.

Although the load balancer in most cases has acted as a dispatcher that is able to handle transparent fail-over occurring on cluster servers, a load balancer easily becomes the single point of failure in a cluster environment. Once the load balancer loses functions under any reasons, the entire system would fail. Such a risk has been eliminated in clustering JBoss with a smart client proxy. The JBoss server instance forces the clients to download stub codes before sending requests. As a result, the client knows a list of available nodes in the cluster family and is given pre-selected load-balancing policy. The smart clients will according to the obtained information send its request to the corresponding server instance. Such an implementation is highly coupling the relation of the client and the load balancer i.e. the dispatcher. Even if the dispatcher fails, it usually means the client code has failed and will not complain [44]. The built-in load-balancing policies in the current JBoss 4.0 version are:

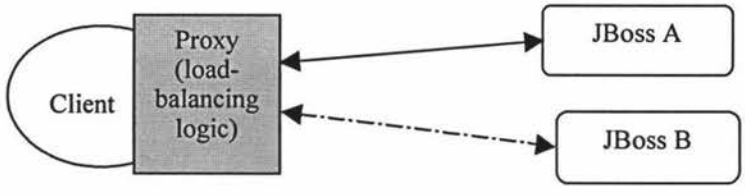


Figure 7: Load Balancer Implementation in JBoss

- *Round robin load-balancing policy*
- *Random robin load-balancing policy*
- *First available load-balancing policy*

} See section 2.3.2.2 for detailed explanations

- *First available identical all proxies*: that is a similar policy to first available load-balancing policy. The first available target node will be selected randomly. But for first available load-balancing policy, each proxy can select its own preferred target node. For this policy, all proxies in the same family share the selected node, i.e. the same family proxies will always communicate to the same node until it dies. After that a new target node will be randomly selected to replace its functions [43].

This study will compare all these built-in load-balancing policies to see if there are significant changes in the final performance of JBoss.

3.2.4.3 Transaction Commit Options in JBoss

Keeping enterprise data in a consistence state under any kinds of transaction modes is critical for any application design. The benefit of a J2EE application server is that it uses containers to handle low level multi-thread processing and resource sharing etc. In most cases the business logic or transaction is managed at EJB level and the business data is stored in entity beans. How the EJB containers handles transactions for entity beans and keep its state consistency in JBoss is important.

As defined in the EJB specification, the application developer doesn't care too much about controlling concurrent access to the entity beans instance since the container has synchronized all access to the same entity bean instance. The EJB specification specifies three 'commit options', and one more D option has been added in the application server JBoss, all these options are available for CMP entity beans [44, 45]:

- *Commit option A*, with this setting, the container is the only access object that can modify the bean state in the database. So there always has to be up-to-date instances in the cache for ready use.
- *Commit option B*, this is JBoss' default setting for entity beans and is also known as pessimistic locking. The synchronized access to the beans state is controlled by the database, i.e. the bean is locked in a transaction until the transaction commits or rolls back. The container also keeps a ready to use bean instance between transactions.

- *Commit option C*, the bean is locked during transaction just like option B but there is no beans instances in the cache. And the bean is passivated at the end of a transaction. For this option, we only can get the state of beans from database instead of from the cache as in option B.
- *Commit option D*, the option works like option A but with a timeout value to validate the bean in the cache, i.e. the bean instance is considered as invalid once the timeout has elapsed and then is removed from the cache.

Although commit option A and D both benefit from the JBoss cache, both of them only allow the container to be the single access point to the database to manage beans state. These two options are not supported in a clustered environment since more than one container can access the database to modify entity beans state and then keep the data in its own cache. The result leads to some data that is never refreshed in some of the instances' cache, which makes the entity beans state inconsistent. So the application server JBoss introduces the Cache Invalidation Framework (CIF) to overcome this restriction for option A. The CIF can automatically remove the cache bean instance when any of containers invalidate a bean instance in a single JBoss or across a cluster of JBoss instances [44]. In other words, once an entity bean is modified in a node, an invalidation message will be sent automatically to all containers in the same cluster and then this entity bean is removed from all nodes' cache. So once the same entity bean is required again, the data will not be found in the cache and the container will get the up-to-date data from the database.

Both option B and C will allow other actors e.g. other containers or objects to modify the beans state since the synchronized access to the bean state is controlled in the database that prevent two or more server instances or containers concurrently modifying entity beans state. Both of them can be used in a cluster environment. For option B, the container will keep ready instances between transactions but the container is not responsibe to keep these instances as valid. And option C has a stronger requirement to keep consistent data only in a database without cache data support. This option should be the best for a cluster JBoss since it protects against fetching any out-of-date entity beans state from cache.

3.2.5 Performance Test Tool – Jmeter 2.0.3

Jmeter is a pure Java-based tool that is used to measure performance and test behaviour under heavy workloads. It requires a Java JDK run time. The current release

version is v2.1. This project uses version 2.0.3 for test. For installation instruction see Appendix E.

Jmeter was primarily designed for web application load testing but it has expanded to test strength of various types of servers, network or objects under stress and to analyze overall performance. Using Jmeter, we can simulate tests for both static and dynamic content such as HTTP pages, Servlets / JSP, Java Object etc [11].

Jmeter provides a GUI for constructing test plans. The test plan starts with adding a thread group, which is a set of independent threads. Each thread represents as an individual client. A thread can be a HTTP request or FTP request etc that is called a sampler. Different kinds of samplers can be used for testing different kinds of servers e.g. web servers, FTP servers etc. In chapter 6, the configuration and implementation of the HTTP request test plan for the web server Tomcat is presented in detail. After configuring a thread group, listeners are added to monitor the test results. The result can be shown on tables or graphs. The listeners can be attached to a single sampler, and it also can be attached to a thread group to record all samplers result. When testing the performance of a server under stress, all samplers are recorded. In addition, Jmeter also provide many features such as assertion, timer and so on. The timer can simulate realistic think time, as if a real client opens a web page and he/she needs time to read information, filling a form or sending a query etc. Furthermore, Jmeter supports distributed testing. The distributed models are master/slave system, only the Master has a GUI to allocate tasks and collect results from the slaves.

The stress test on Jmeter can be controlled via three main parameters, number of threads, ramp-up period (on seconds) and loop count [11, 51].

- *Number of threads*: basically represents the number of independent clients participating in the simulation test.
- *Ramp-up period* (on seconds): the period to start all the threads for example, if the number of thread groups is 10 and each thread group contains 5 threads on the test plan and the ramp-up period is 2 seconds, then each thread will be delayed 1/25 seconds after the previous one starts. The total 50 threads will start within the 2 seconds.
- *Loop count*: this indicates the number of times the threads should run. Each loop represents the same number of clients repeating the same tasks – sending the same requests to the server to simulate the stress on the application server.

Chapter 4: Benchmark J2EE Application

Summary: This chapter details the entire benchmark J2EE application – MGProject design, including database, EJB and web components design. A brief introduction of the EJB and Servlet design, including the life cycle of the EJBs and Servlets, including the main concerns on designing these components is given. In addition, example coding of the design to explain the components are given. Moreover, how the web component and EJB components interact with each other in the application and how to use XDoclet in the NetBeans IDE to generate JBoss specific deployment descriptors is presented.

4.1 Case Study

One of the tasks in this research is to design a benchmark J2EE application. It will be used for testing the performance of the load balancing mechanism. The J2EE benchmark application ECperf discussed in section 2.2 is the most well-known test application for J2EE application server and EJB container performance testing. The SPECjAppServer2004 benchmark has enhanced Ecperf by modifying access to the application via the web layer to replace the old versions that directly communicate to the EJB. The MGProject application basically combines both design styles but concentrates on the customer domain.

The case study chosen is a small printing company - M&G printing company Ltd. The company consists of one office, one manufactory and a chain of suppliers to provide one-stop service to its customers from design to finishing. Its printing production includes advertisement sheets, journals, catalogues, brochures, books, invitations and package bags. In order to support better management and services, the company needs a computer system to manage its business. The case has a similar business processes as ECperf based on four business domains corporate, customer, manufacturing and supplier. This benchmark application design only focuses on the customer domain that includes a company website for the convenience of its customers. The functions of this benchmark application should include:

- Customers can freely quote prices of various printing materials, such as

advertisement sheets, books / magazines or package bags and so on. Currently, the application is designed for advertisement sheets only but it can be extended for quoting other materials.

- A customer can make one or more online orders if the customer is satisfied with the quoting prices. Each order should have an appropriate detail description and a unique ID.
- New customers are required to register first before he/she makes an order. Each customer has a unique ID.
- The customer may use the customer ID and order ID to check a particular order status or cancel the order online later.
- The registered customers can update their personal information and check all outstanding orders at any time.

The main functions of the benchmark J2EE application system can be illustrated in a use case diagram in figure 8. Each use case represents a single task / event that displays web pages as a button or a link label or a web page. The "include" link shows one task should always invoke another task event, such as when a customer clicks the link label "Make a New Order", another web page "Quote Price" should always display. The "extend" link represents one single task, which may invoke another task event. For instance, if a customer satisfies the quoting price and decides to make an order, if the customer is an existing customer with registered custID, then the event "Register New Customer" will not be invoked. But if the customer is a new customer, this event should be invoked too.

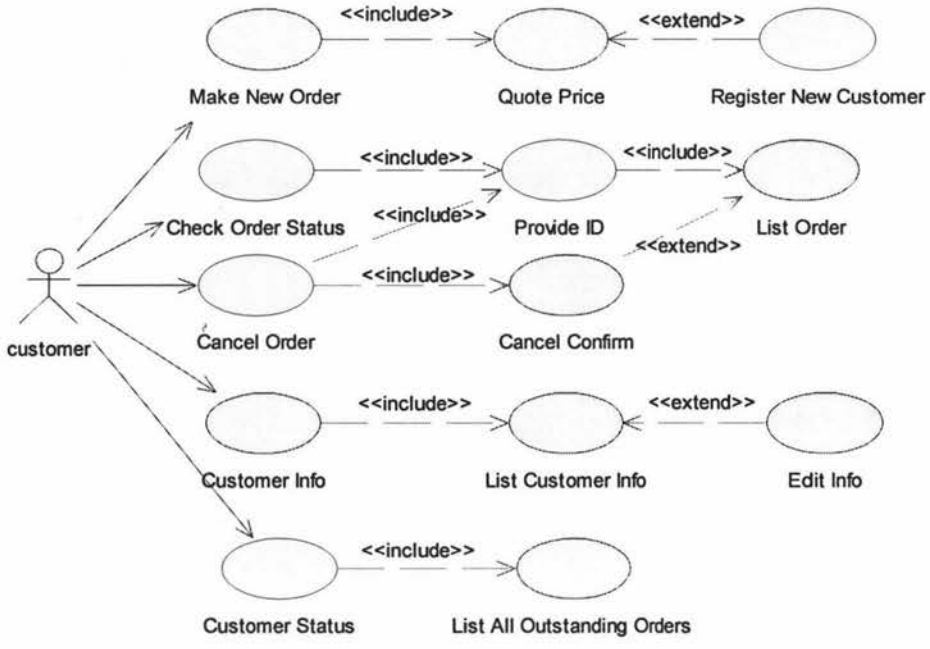


Figure 8: Use Case Diagram of the Benchmark Application

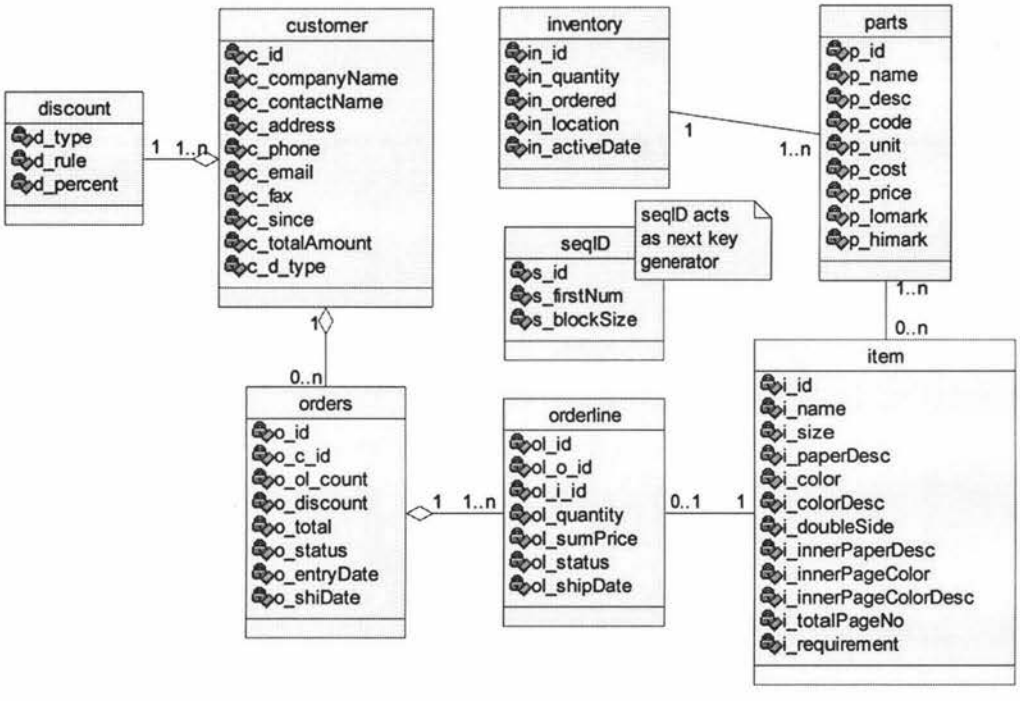


Figure 9: Class Diagram Showing Database Design

4.2 Database Design

The application design focuses on the customer domain as shown in figure 8. The main task in the customer domain is to manage orders, e.g. making order, checking order status and cancel order etc. All relevant persistent data are stored in tables in a database – named *mgDB* in this application.

According to the system requirements, eight tables store the business data. The database table's fields are shown as attributes of a class object in the entity beans' class diagram (see figure 9), this is an ideal diagram to reflect all fields of the tables in the *mgDB* database and relationships of tables.

Although the application design only focuses on the customer domain for online customers' behaviors, the *mgDB* design considers the entire business processes and can be extended and applied to the other domains. Some fields of the table are not used in the current application, but it may be used in another business domain later. For example, *p_lomark* and *p_himark* in the parts table refers to the quantity of a specific part that has to be stored in the inventory at least more than the lowest *p_lomark* value but less than *p_himark* value, which determines when and how many parts have to be purchased from suppliers. And in some tables, the fields can be accessed by more than one domain. For instance, *ol_status* in the orderline table, the status of the online order is "ordering" when a customer just ordered an order. This field will be modified in the manufactory domain when the order is "Processing" or "Shipped".

The *mgDB* is a relational database so each table has a primary key to identify each record and a foreign key maintains the relationships within tables. The primary key of most tables (except table discount and *SeqidBean* that have String values for the primary key) is automatically incremented numbers that starts with value 1. The name of fields in each table is meaningfully named. The information stored in each table is as follow:

- Customer – refers to all customers (i.e. online customers & non-online customers) of the MG Company. The table is used to store general customer's details, including contact details and discount type. The discount type depends on the total amount of all orders (on the field c_totalAmount) that the customer has purchased since the customer registered. The foreign key *c_d_type* maintains the relationship between customers and discount tables.
- Discount – is use to store the company discount policies for its customer.

Currently these are four levels of discount policies in the application, i.e. gold customer, silver customer, first time customer and general customer. Different policies levels have various discount value. For example, each first time customer always gets 5% discount when the customer registers. For long-term customers, if the accumulated purchase amount exceeds $300,000, the customers will automatically become silver customers that are able to get 18% discount for their future orders.

- Orders – refers to all online and non-online orders related to one customer. That means every time each customer has only one order. So there is a foreign key field *o_c_id* that points to the customer table to find a particular customer for the order. But each order may have one or certain orderlines. The current application has an orderline table to store information. The orders table mainly records the order date and shipped date, the order status, the total order price and the number of online orders i.e. orderline.

- Orderline – records online order information including price, quantity, the status of the online order and the shipped date. Since the main task in customer domain is to manage orders of printing production of MG Company, there are three tables orders, orderline and item to handle the order information that can be managed and extended. So there are two foreign keys *ol_o_id* and *ol_i_id* in the orderline table linked to the two tables orders and item.

- Item – is the table that keeps the printing production requirements. The table records details about printing material (e.g. advertisement sheets or books etc), size of the production, prefered paper, color requirement and so on. Some fields that describe inner pages like *i_interPaperDesc*, *i_totalPages* etc are used for recording the requirements of printing books / magazines that are set to NULL for the current application.

- Inventory – is mainly used for storing the supplied materials for the manufactory and office. The information includes the quantity of supply, the active date and store location, since the manufactory and the office are usually in different location. The location field is required. The primary key in_id is also as a foreign key link to the parts table.

- Parts – is the table to describe the detail about each supply, including its name,

production code and unit measurement, purchase price, selling price, lowest and highest storage. This is the main table to provide the price quote for the printing materials. For example, usually the purchased paper is stored in the inventory in original size 1194mm * 889mm. The paper always has to be cut to fit the finishing advertisement sheets size, e.g. 456mm * 312mm. So the program will calculate the total original paper need for use and what the price is and so on.

- Seqid – the table is used for storing the instance ID value, which is required by the program to create an instance of entity bean i.e. customer, orders, orderline and item. The primary key *s_id* directly refers to these entities, such as "customer", "orders" etc. The field *s_firstNum* refers to the next available ID number and *s_blockSize* is the sequence block size. This is the critical table for all new instances of customer, orders, orderline and item entity beans have to concurrently access. Details are presented in section 4.3.1.

The definition of the above eight tables is created via a text editor and saved in a file *mgDB.sql* (see Appendix F) so that the database can be easily created during benchmark testing.

4.2.1 Database Implementation

The application uses JBoss as the J2EE application server. The built-in database Hypersonic can be run as an embedded service within JBoss. So in JBoss there is a corresponding data source set up in the JCA (J2EE Connector Architecture provides connectivity between application servers and existing Enterprise Information System [42]) i.e. the default datasource JNDI name "*java:/DefaultDS*" refers to the hypersonic database. Hypersonic is convenient but in the real world, most enterprises have other existing database systems, which are vital parts in the enterprise application. In some cases, the enterprises desire a new J2EE application is able to integrate with the existing database, not just the data itself, but also including all database features such as stored procedures, trigger etc.

Another attractive feature for JBoss is that it can easily port a J2EE application to a third-party database system such as MySQL, Oracle, PostgreSQL etc. To port a third-party database system is also a challenging task in this study. This study uses the PostgreSQL database.

The standard API JDBC is used to connect between the JBoss AS and PostgreSQL.

JDBC support single connections, i.e. the connection need to be created every time when JBoss AS interacts with the database. It also supports a connection pool, which is a collection of database connections maintained and managed by the JBoss AS. The connection pool is a DataSource object that enhances the performance of running an application because the process of establishing a connection to a database is usually expensive and slow [4]. The concept has been used in this application to improve performance.

4.3 J2EE Components Design & Implementation

As shown in figure 1 in chapter 2, the J2EE components include the web components and EJB components, which is the key to successfully run a J2EE application. Web components design includes the user interfaces and the EJB components that manage business processes logic. Usually the business logic is invoked via a user interface event. Therefore, the EJB components are designed first since the business methods can be called immediately from the web components.

4.3.1 EJB Design

The enterprise Java bean (EJB) design is the most important part of a J2EE application, which implements the business logic of an enterprise application. The J2EE architecture in figure 1 shows there are three kinds of EJBs --- entity beans, session beans and message driven beans. All EJB run in an EJB container that is a runtime environment within the J2EE application server. The Message driven beans are invoked by an incoming message; it is not used in the benchmark application. Therefore, this benchmark application focuses on entity beans and session beans design. The entity beans are used for storing business persistent data and the session beans handle business processes logic.

Different from the original object class, each EJB has two interfaces - home interface and component interface and the bean class. The home interface is used to create, remove and find EJB instances. The component interface exposes EJB business methods to clients. The bean class is used to implement the methods defined on the interfaces. The clients never access beans directly. Access needs to be gained through a container and then in turn invoke the beans methods. Each EJB has a unique JNDI name (standard interface to Naming and Directory Services for enterprise applications) when it is deployed into a J2EE application server, which provides an "address" to the EJB container to lookup and access bean services. The default JNDI for any EJB is mapped

into *java:comp:/env/ejb* directory.

On the other hand, each EJB also has two sets of interfaces, local and remote interfaces. The local interface can only be accessed by the bean located on the same JVM. It was first introduced in the EJB 2.0 specification to improve performance. The remote interface is a RMI interfaces to allow beans to be location independent. But the calling semantics are different, remote interfaces pass parameters inefficiently using call-by-value and local interfaces use call-by-reference.

Although each EJB can be assigned both local and remote interfaces during design, the EJB specification recommends each EJB have one suitable interface. Therefore, the kind of interface assigned to an EJB is decided during design. In addition, as stated before, one of the main features of a J2EE application is the n-tiered architecture. This characteristic (see Figure 10) exists in this MGProject benchmark application. As a result, Entity beans in the MGProject application have a local interface and session beans have a remote interface, since the entity beans are accessed only by the session beans or other entity beans that are in the same JVM. The session beans need a remote RMI interface to communicate with Servlets across the network.

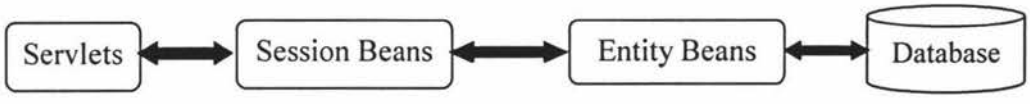


Figure 10: The Strict Communication in This Benchmark Application

4.3.1.1 Entity Bean

Entity beans represent business persistent data mapping to one row of a database table. For example, each customer has their own name, address, phone number and so on, which is stored in the customer table in PostgreSQL. The state of an entity bean is persistent and transactional. It can be shared among different clients. There are two ways to manage the persistence of Entity beans - by the EJB container or by the bean itself. The Bean Managed Persistence (BMP) means the developers have to write code for *ejbLoad(), ejbStore()* and other methods that are defined on the *javax.ejb.EntityBean* interface to manage both database connections and the change to the beans' state. But Container Managed Persistence (CMP) will directly map the bean fields with the table fields in the deployment descriptor. Therefore, any change in entity beans' state will be auto saved to the database. Nowadays, CMP is recommended since it is quite efficient and less error prone. In this application, all entity beans persistence is managed via the JBoss EJB container.

The EJB container manages bean persistence via a deployment descriptor (see example in Figure 11) that are XML documents to describe the detail settings of each entity bean, which maps the abstract persistent data schema to a physical schema of the underlying tables in the database. The sample deployment descriptor maps the entity *OrdersBean* to the *orders* table in PostgreSQL. The deployment descriptor can be created manually or generated via a tool. The information is read by the EJB container at deployment time.

Although CMP entity beans are managed automatically via the EJB container, it is necessary to know the interactions among a client (web components or other EJB), EJB container and CMP bean in detail. Figure 12 shows the details of a CMP entity bean [4, 5].

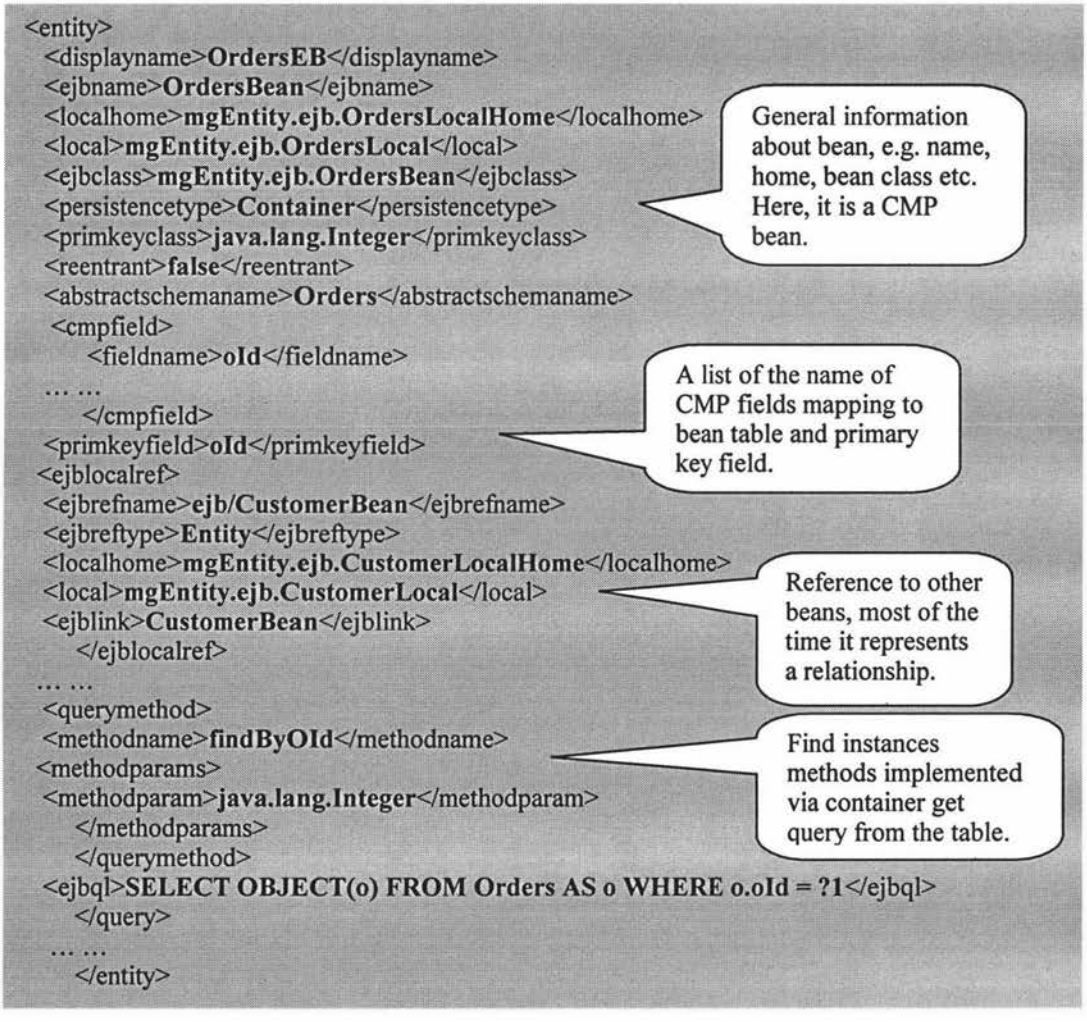


Figure 11: Example Deployment Descriptor of Entity Bean

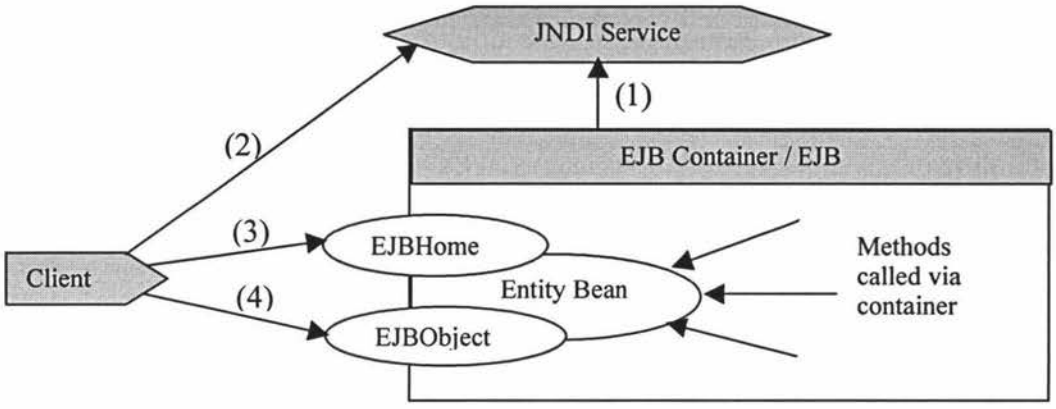


Figure 12: Details of CMP Bean

The sequence of steps (see numbering on Figure 12) for managing CMP entity beans are [4, 5]:

(1) The EJB container firstly registers all deployed EJB with JNDI service.

(2) The client looks up the home interface of deployed EJB via JNDI service.

(3) The client uses the home interface to create a new instance of EJB or find an instance of EJB via calling *create* or finder methods e.g. *findByPrimaryKey*.

(4) The client calls the business methods defined on the component interface, the methods implement at entity bean class that are invoked via the EJB container.

Another feature of the EJB container is that it is able to automatically manage the relationship among entity beans – Container-Managed Relationships (CMR). The relationship can be one-to-one, one-to-many or many-to-many. Figure 9 (section 4.1) shows this relationship in this application. For example, each "customer" may have one or more "orders". But each "orders" should belong to only one "customer". The example shows a one-to-many relationship between "orders" and "customer" entity beans in this case.

However, CMR is applicable only to CMP beans and not for BMP beans. So if we require the EJB container to manage the beans relationship, the beans should be CMP beans with a local interface.

For CMP and CMR bean, the deployment descriptors are documents used by the EJB container to generate the data access calls to the underlying database at deployment time. The standard EJB deployment descriptor *ejb-jar.xml* contains only Sun Microsystems-specific deployed EJB properties. For different vendor's J2EE server, there is also a

vendor-specific deployment descriptor. For JBoss, two deployment descriptors *jboss.xml* and *jbosscmp-jdbc.xml* have to be configured correctly to ensure no errors for deploying EJB into JBoss.

The following example (*OrdersBean*) is given to present detail about CMP and CMR entity bean design including home and component interface and bean class for *OrdersBean* with only a local interface.

The Home Interface

The local home interface must extend *javax.ejb.EJBLocalHome* (the remote interface extends *javax.ejb..EJBHome*), which enables the clients to perform the following actions via the EJB container [4]:

- create an instance of entity bean
- find an entity bean object
- remove an instance of entity bean

Therefore, the home interface only defines create, finder and remove methods. Since each entity bean has a unique ID i.e. a primary key identifies the entity bean. The *create* method should accept parameters – primary key field and other data fields. The create method can be overloaded in the home interface. Three types of finder methods can be applied on the home interfaces, i.e. *findByPrimaryKey, findBy<field_name>* and *findAll*. Figure 13 shows the sample code for entity "orders" with only a local interface.

```
/**
 * This is the local-home interface for Orders enterprise bean.
 */
public interface OrdersLocalHome extends javax.ejb.EJBLocalHome {

    mgEntity.ejb.OrdersLocal findByPrimaryKey(java.lang.Integer key)  throws
                javax.ejb.FinderException;

    public mgEntity.ejb.OrdersLocal create(java.lang.Integer oId, java.lang.Integer oOlCount,
    java.math.BigDecimal oDiscount, java.math.BigDecimal oTotal, java.lang.String oStatus,
    String oEntrydate, String oShipdate, mgEntity.ejb.CustomerLocal oCId) throws
    javax.ejb.CreateException;

    java.util.Collection findByOId(java.lang.Integer oId) throws javax.ejb.FinderException;

    ... ...
}
```

important method to find an instance

related to Customer bean

findBy<field_name> always return a collection of bean objects

Figure 13: Example of Local Home Interface Design for Entity Bean

The Component Interface

The component interface exposes business methods called by clients. The local interface extends javax.ejb.*EJBLocalObject* (if it is a remote interface should extend *javax.ejb.EJBObject*). For CMP bean's component interface, there is a long list of abstract methods *get<field_name>* and *set<field_name>*. The abstract methods will be mapped to the table fields in the database via XML deploy descriptor documents by the EJB container. The NetBeans IDE separates the component interface into two parts, one just defines the interface is local or remote without methods and another business interface part contains the declaration of all exposed business methods.

```
package mgEntity.ejb;
/**
 * This is the local interface for Orders enterprise bean.
 */
public interface OrdersLocal extends javax.ejb.EJBLocalObject,
mgEntity.ejb.OrdersLocalBusiness {

}
```

Just defines a local interface

```
package mgEntity.ejb;
/**
 * This is the business interface for Orders enterprise bean.
 */
public interface OrdersLocalBusiness {
    public abstract java.lang.Integer getOId();
    public abstract java.lang.Integer getOOlCount();
    public abstract void setOOlCount(java.lang.Integer oOlCount);
    public abstract java.math.BigDecimal getODiscount();
    public abstract void setODiscount(java.math.BigDecimal oDiscount);
    ... ...
    public abstract java.util.Collection getOrderlineBean();
    public abstract void setOrderlineBean(java.util.Collection orderlineBean);
    ... ...
}
```

Directly get from or set to the underlying database

It is a foreign key field that represents a relationship between beans.

Figure 14: Example of Local Component Interface Design for Entity Bean

The Bean Class

All entity bean class implements the *javax.ejb.EntityBean* interface. The methods are already defined in the interface [4, 5]:

- *setEntityContext*: setting associated entity context e.g. identifying caller, obtaining the primary key associated instance, storing instance variables etc
- *unsetEntityContext*: called before removing an instance. That is a place to release resource that associated with the instance
- *ejbActivate*: restore the bean information from the database when the bean instance is referenced
- *ejbPassivate*: release any allocated resource when the number of entity beans exceeds certain threshold of bean pool
- *ejbLoad*: synchronize bean's state by loading from the database
- *ejbStore*: synchronize bean's state by storing to the database
- *ejbRemove*: remove the bean instance

For BMP, the developers need to be concerned about the above methods. But it is not the case for CMP; all are under control of the EJB container automatically. And other two methods *ejbCreate* and *ejbPostCreate* will be called when a client creates a new instance and calls the *create* method on the home interface. Finder methods like. *ejbFindByPrimaryKey* or *ejbFindBy<field_name>* will be coded by the developer only for BMP bean and the container will call then when a client invokes finder methods.

The bean class has method implementations that are declared on home and component interfaces but for CMP bean, it excludes the container managing methods e.g. get<field_name>, set<field_name>, ejbFindByPrimaryKey, and ejbLoad etc. The example bean class from the MGProject application is shown in figure 15.

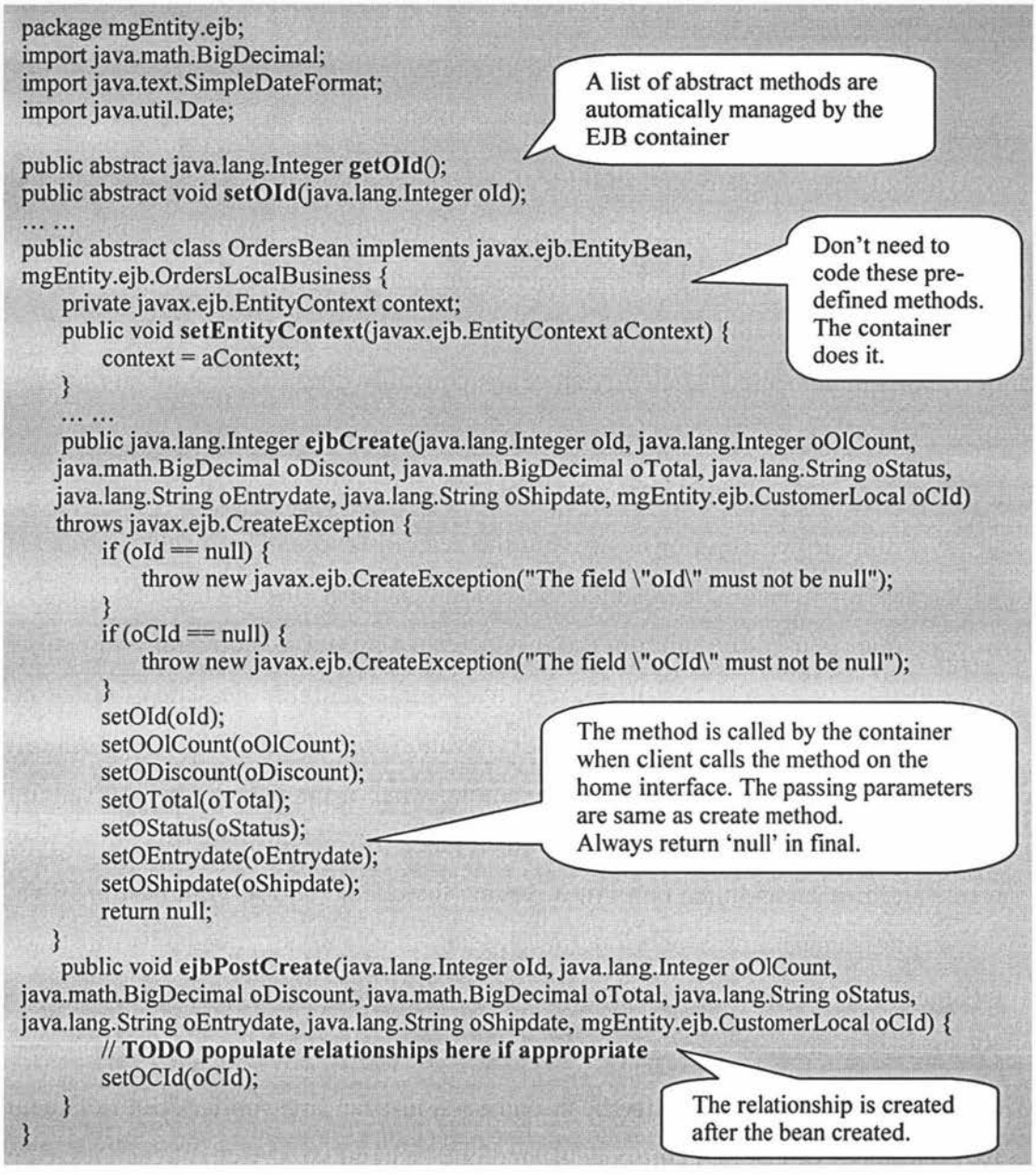


Figure 15: Example of Bean Class Design for Entity Bean

4.3.1.2 Session Bean

The session beans model business processes logic and rules e.g. performing calculations, searching data etc. Unlike the entity beans, the session beans typically have the following distinct characteristics [4, 5]:

- Representing a conversation between a single client and a server
- The bean cannot be shared among multiple users or multiple threads at the same time.

- The state of bean is not persistent so it cannot represent data in a database, but it can perform a task to update data.
- No relationship needs to be maintained among beans.

Two types of session bean exist in the EJB container, the stateless session bean and the stateful session bean. The difference between them is that the stateful session bean is responsible for keeping the state for a particular client during a session. But the stateless session bean does not maintain any state associated with any client. So the stateless session bean is implemented efficiently since the container does not care about the state of bean.

The instance pool in the EJB container improves the performance for beans reuse since it saves time for creating or destroying the bean instances. The mechanism is the most efficiency for managing the stateless session beans because of the lack of state. The instance pool is used to maintain the instances of stateful session beans and entity beans, but just the types of bean without identity. For maintaining the state of the bean, the EJB container performs passivation and activation to store and fetch the state of beans from secondary storage [4, 5]. For example, when a client calls back a stateful session bean from the instance pool, the bean instance just performs activation to load the saved state of bean and then continues serving the client. The serving bean instance may not be the same bean as previous one. Meanwhile, another mechanism instance caching maintains the particular bean instance but for the stateful session beans and entity beans [4].

The implementation of the entity bean and session bean are similar. Both of them have two interfaces (home and component) and one bean class. A client cannot create an instance of EJB directly or access the EJB object without the EJB container help. Only via the container, the deployed session beans register their home interface into the JNDI service. Then clients look up the JNDI service to get the home object to create or find a particular bean instance, and then perform business processes via invoking the methods defined on the component interface.

In addition, as stated before, each EJB has at least one interface, local or remote interface. Usually the session beans have to serve a remote client that may be a web component, another EJB or an application client. So the session beans are assigned a remote interface in most cases.

4.3.1.2.1 Stateless Session Bean

A stateless session bean is similar to the above entity bean design that requires two interfaces and one bean class and a deployment descriptor but is simpler.

The Home Interface

The session bean with RMI remote interface extends *javax.ejb.EJBHome* on the home interface. It is only for creating and removing an instance of bean without finder methods. Usually there is only one method *create()* without any parameter since no ID is related to it. While a client requires an instance to invoke the *create()* method, the container will pull one created instance from the instance pool and the instance returns to the pool after the client completes a task. For sample code from the MGProject application for the stateless session bean home interface design see figure 16.

The Component Interface

This is main location to expose business logic and rules. The remote interface extends *javax.ejb.EJBObject* with business methods. All methods on the remote interface should include a throws clause *java.rmi.RemoteException*. The exception is thrown when a remote invocation fails because of network failure or protocol errors etc. Figure 17 shows the component interface design for the stateless session bean. The interface also is separated into two parts, a remote interface and business interface.

The Bean Class

The stateless session bean implements the *javax.ejb.SessionBean* interface. The methods already defined in this interface include *setSessionContext(), ejbCreate(), ejbActivate(), ejbPassivate()* and *ejbRemove()*. In fact, the methods *ejbActivate()* and *ejbPassivated()* are not applicable to the stateless session bean since the bean does not maintain the state. The bean class will implement all methods declaring on the business interface using business rules. The sample implementation of a stateless session bean *orderSessionBean* from the *MGProject* application is shown in figure 18.

```
package mgSession.ejb;
/**
 * This is the home interface for orderSession enterprise bean.
 */
public interface orderSessionRemoteHome extends javax.ejb.EJBHome {

    mgSession.ejb.orderSessionRemote create()  throws javax.ejb.CreateException,
java.rmi.RemoteException;
}
```

No ID relates to a stateless bean, so no parameters are passed for the create method. In addition, the method cannot be overloaded.

Figure 16: Example of Remote Home Interface for Stateless Session Bean

```
package mgSession.ejb;
/**
 * This is the remote interface for orderSession enterprise bean.
 */
public interface orderSessionRemote extends javax.ejb.EJBObject,
mgSession.ejb.orderSessionRemoteBusiness {
}
```

Define the interface is remote.

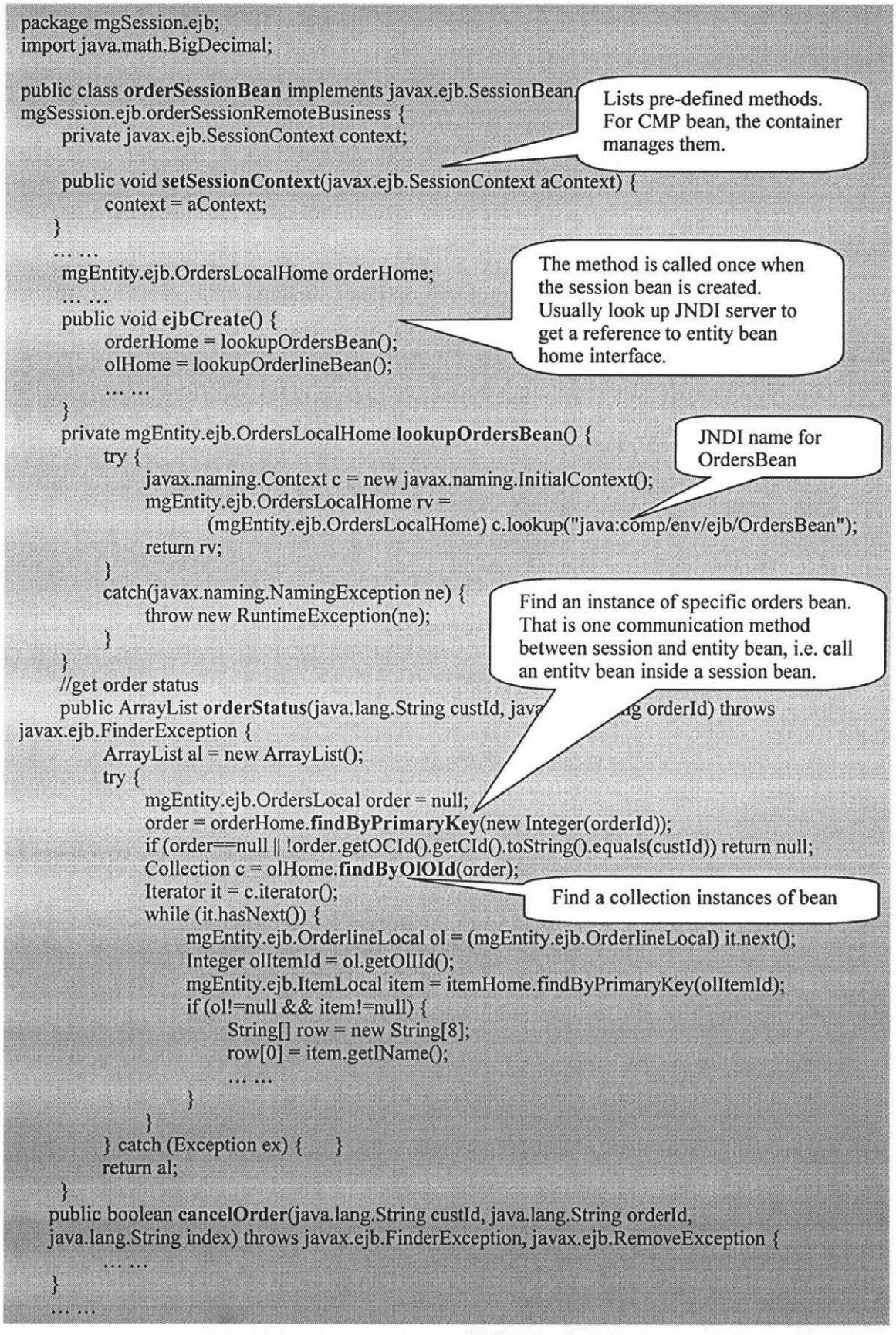
```
package mgSession.ejb;
import java.util.ArrayList;
/**
 * This is the business interface for orderSession enterprise bean.
 */
public interface orderSessionRemoteBusiness {
    ArrayList orderStatus(java.lang.String custId, java.lang.String orderId) throws
javax.ejb.FinderException, java.rmi.RemoteException;

    boolean cancelOrder(java.lang.String custId, java.lang.String orderId, java.lang.String index)
throws javax.ejb.FinderException,javax.ejb.RemoveException,java.rmi.RemoteException;

    ArrayList getCustStatus(int custId) throws javax.ejb.FinderException,
java.rmi.RemoteException;
}
```

To place business methods here that is behaviour to process business rules.

Figure 17: Example of Component Interface for Stateless Session Bean




```
package mgSession.ejb;
import java.math.BigDecimal;

public class orderSessionBean implements javax.ejb.SessionBean,
mgSession.ejb.orderSessionRemoteBusiness {
    private javax.ejb.SessionContext context;

    public void setSessionContext(javax.ejb.SessionContext aContext) {
        context = aContext;
    }
    ... ...
    mgEntity.ejb.OrdersLocalHome orderHome;
    ... ...
    public void ejbCreate() {
        orderHome = lookupOrdersBean();
        olHome = lookupOrderlineBean();
        ... ...
    }
    private mgEntity.ejb.OrdersLocalHome lookupOrdersBean() {
        try {
            javax.naming.Context c = new javax.naming.InitialContext();
            mgEntity.ejb.OrdersLocalHome rv =
                (mgEntity.ejb.OrdersLocalHome) c.lookup("java:comp/env/ejb/OrdersBean");
            return rv;
        }
        catch(javax.naming.NamingException ne) {
            throw new RuntimeException(ne);
        }
    }
    //get order status
    public ArrayList orderStatus(java.lang.String custId, java.lang.String orderId) throws
javax.ejb.FinderException {
        ArrayList al = new ArrayList();
        try {
            mgEntity.ejb.OrdersLocal order = null;
            order = orderHome.findByPrimaryKey(new Integer(orderId));
            if (order==null || !order.getOCId().getCId().toString().equals(custId)) return null;
            Collection c = olHome.findByOlOId(order);
            Iterator it = c.iterator();
            while (it.hasNext()) {
                mgEntity.ejb.OrderlineLocal ol = (mgEntity.ejb.OrderlineLocal) it.next();
                Integer olItemId = ol.getOllId();
                mgEntity.ejb.ItemLocal item = itemHome.findByPrimaryKey(olItemId);
                if (ol!=null && item!=null) {
                    String[] row = new String[8];
                    row[0] = item.getIName();
                    ... ...
                }
            }
        } catch (Exception ex) {    }
        return al;
    }
    public boolean cancelOrder(java.lang.String custId, java.lang.String orderId,
    java.lang.String index) throws javax.ejb.FinderException, javax.ejb.RemoveException {
        ... ...
    }
    ... ...
```

Figure 18: Example of Bean Class for Stateless Session Bean

4.3.1.2.2 Stateful Session Bean

The stateful session bean design is very similar to stateless session bean design except the EJB container assigns a unique ID for each stateful session bean instance at creation time. But this ID is not exposed to a client i.e. the client is unconscious of the existing ID. The client can use *isIdentical* methods on the *javax.ejb.EJBObject* to identify whether two session bean instances reference to the same bean [4]. Therefore, a stateful session bean may overload the method *create()* in the home interface. In addition, during the life cycle of the stateful session bean, the passivation and activation mechanism will be used for maintaining state by the container.

Since the design style of the stateful session bean is similar to stateless session bean design, it is discussed briefly here.

Calling a stateful session bean is different from non-HTTP clients and HTTP clients. There would be no problem for non-HTTP clients to reference the stateful bean during its life cycle since the EJB container maintains the bean state. However, HTTP is a stateless protocol, which means it is hard to maintain the session for HTTP clients. One solution used in the Servlets of the MGProject application is to use the *javax.servlet.http.HttpSession* object, which provides a way to apply the *HttpSession* to the stateful session bean [5]:

- Firstly, get a reference to the stateful session bean and store it as an attribute in the HttpSession object, see

```
HttpSession session = request.getSession(true);
Session.setAttribute("cart", cartBean);
```

Figure 19: Set HTTP Session for Stateful Session Bean

Since mostly we use the stateful session bean to keep a session information like shopping cart. Here, "cart" is storage object we defined in the stateful bean and "cartBean" refers to the name of bean class.

- Secondly, we need to access the session bean object via the reference from HttpSession object. In figure 20, "Cart" is a remote stateful session bean.

```
HttpSession session = request.getSession(true);
Cart cartBean = (Cart) session.getAttribute("cart");
```

Figure 20: Get HTTP Session for Stateful Session Bean

4.3.2 EJB Implementation in the MGProject Application

The benchmark J2EE application is named *MGProject*. All EJB source files and relevant documents are under *MGProject/MGProject-EJBModule* directory. The source codes of EJB are classified into two main packages to hold different types of EJB, i.e. *mg.ejb.EntityBean* contains only source file of entity beans and *mg.ejb.SessionBean* holds session beans source codes. Under the source package, there are two more packages that contain special entity beans and session beans needed for the program. One is *mg.ejb.helper* that holds the *SeqidBean* entity bean and SeqidSessionBean session bean to generate the next key number for other entity beans i.e. *customer, orders, orderline* and *item*. Another package *mg.ejb.mgPolicy* contains a timer session bean designed load-balancing policies.

4.3.2.1 Entity Bean

NetBeans 4.1 makes writing J2EE application easier. Once the entity bean tables are defined in the database, it can directly generate CMP entity bean classes from the database including home and component interface and bean classes, but it separates the local component interface into a local interface and business interface. So each bean has four Java classes related to it. To implement seven entity beans i.e. *customer, discount, orders, orderline*, *item, inventory* and *parts* that are related to the tables in the PostgreSQL, there are a total of 28 classes under *mg.ejb.EnityBean* directory. The function of each entity bean is given in the database design, and entity beans design is discussed in section 4.3.1.1.

The advantage of a CMP bean is that the bean relies on the EJB container to do the mapping job between the fields of bean and underlying entity table. The task is done automatically in NetBeans IDE via standard deployment descriptor *ejb-jar.xml*. However, the application will run on JBoss that requires JBoss specific deployment descriptors to support i.e. *jboss.xml* and *jbosscmp-jdbc.xml*. The *jboss.xml* contains the information of deployed entity beans and The *jbosscmp-jdbc.xml* describes the fields mapping and relationships for these beans. The JBoss deployment descriptors need to be created for these entity beans. The XDoclet [47] tags inside the Javadoc can generate JBoss specific deployment descriptors. With experience this can be done. Figure 21 shows sample XDoclet tags for entity bean *OrdersBean* that in turn generate JBoss specific deployment descriptors.

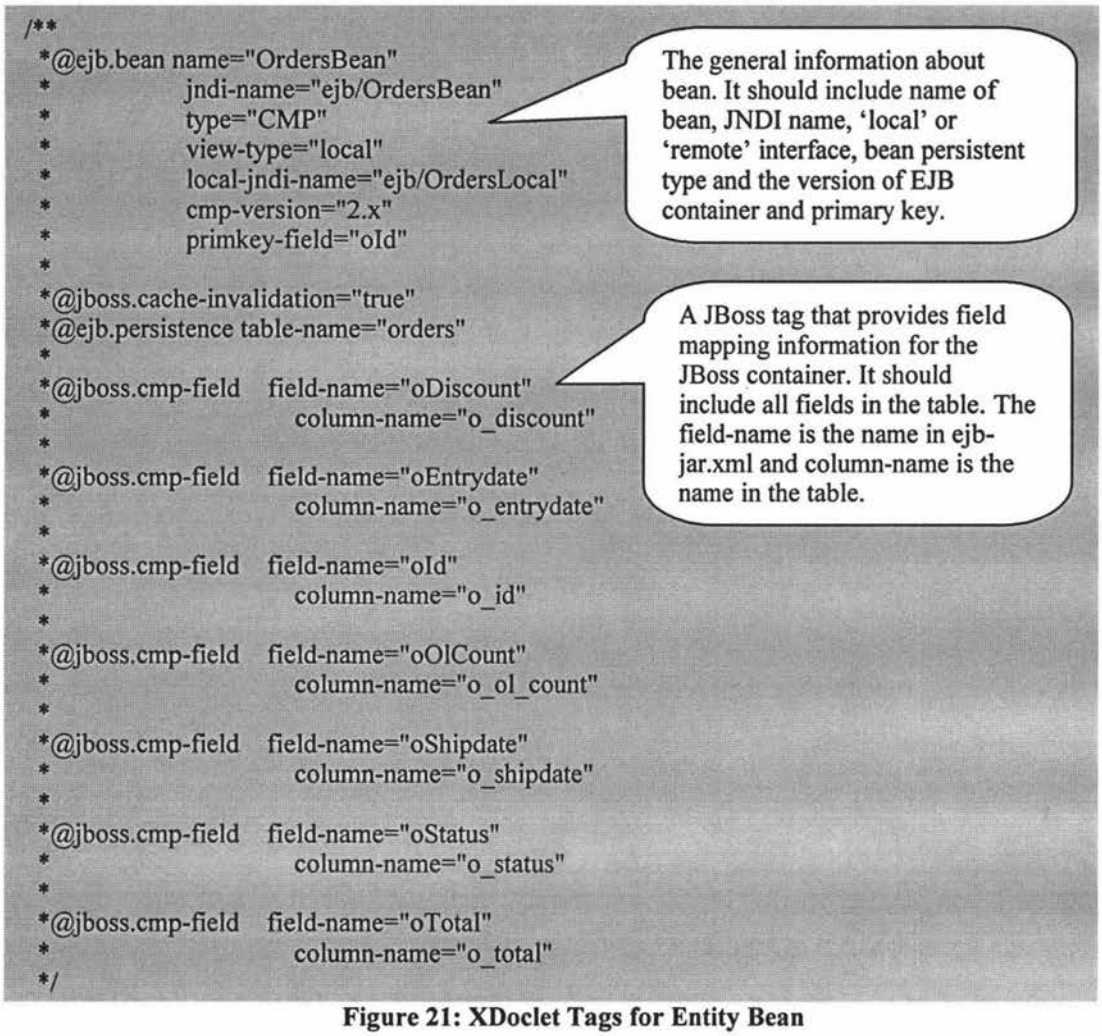


Figure 21: XDoclet Tags for Entity Bean

The *SeqidBean* is a special entity bean designed since each entity bean creation needs an identity primary key. So we need an entity that can remember the next available number that can be used for creating an entity without key clash. The *SeqidBean* was designed for this purpose. The entity *SeqidBean* stores the next key value for four business entities in the application, i.e. *customer, orders, orderline* and *item* entities. However, the EJB container has concurrent control for accessing the entity bean to keep the data persistence. That means every time only one single process or thread is allowed to access to *SeqidBean* to get the next key value, which leads performance drops. In addition, the application server allocates a process or a thread a certain amount of waiting time e.g. 30000ms for processing. A "TimeOut" exception will be thrown when too many processes wait. Therefore, this design introduces a stateless session bean *SeqidSessionBean* and *SequenceBlock* object (this is a Serializable object to store sharing data) to temporarily hold a block (block_size = 100 in the benchmark application) as a next key generator to serve a number of clients. The mechanism is illustrated in figure 22 (for detail design sees source codes).

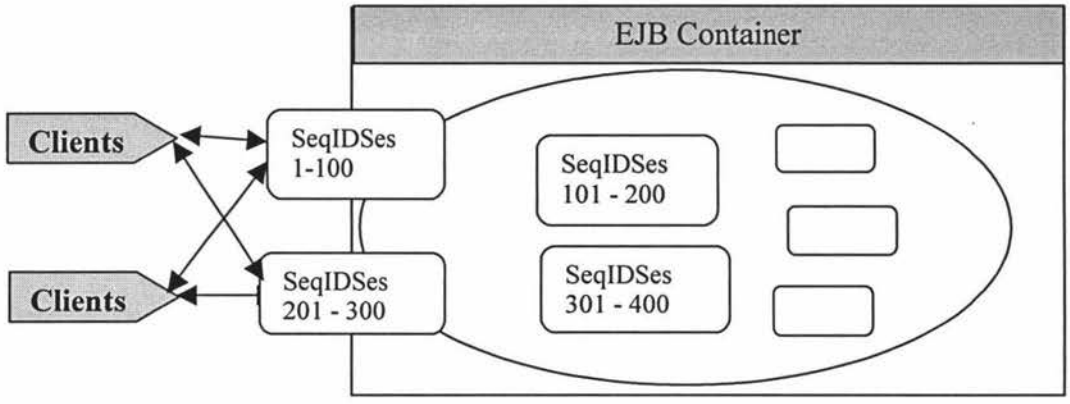


Figure 22: The Next Primary Key Generator System

Figure 22 shows the next primary key generator system. The object *SequenceBlock* stores a block of numbers for each entity *customer, orders, orderline* and *item*. The EJB container uses an instance pool policy to create a number of stateless *SeqidSessionBean* beans, which in turn invoke the *SequenceBlock* object to get a block of available numbers and store it in a *HashMap* variable (a pair of entity type and *SequenceBlock* object). The mechanism enables each session bean to store a block of unique available key numbers when the beans are deployed into the container. When a client requires the next key value, the container will pull a bean from the instance pool and allocate an available key number. After the client gets a number, the instance returns to the pool for reuse next time. The instance of *SequenceBlock* remembers a new next available number for the entity. The generator ensures each client can get a unique key value but not guarantee the created entities have a sequential key value in the database.

4.3.2.2 Session Bean

The session bean design depends on the business processes. According to the system requirements i.e. processing online transaction in the customer domain, four stateless session beans and one stateful bean are created. The functionality for these beans includes quoting price, registering new customers, making new orders, canceling an order and so on. Each session bean assigns a remote interface. The function of each session bean methods (each method has a meaningful name) is defined at the remote interface that the clients (Servlets in my case) can call. The detail about each bean design i.e. home and component interfaces and bean class design is similar to that discussed in section 4.3.1.2. NetBeans generates a total of 20 source classes for these session beans. Figure 23 illustrates the designed methods on each session bean and the entity beans they reference. Since each entity bean has a local interface, it is convenient for session beans to access them within the same JVM.

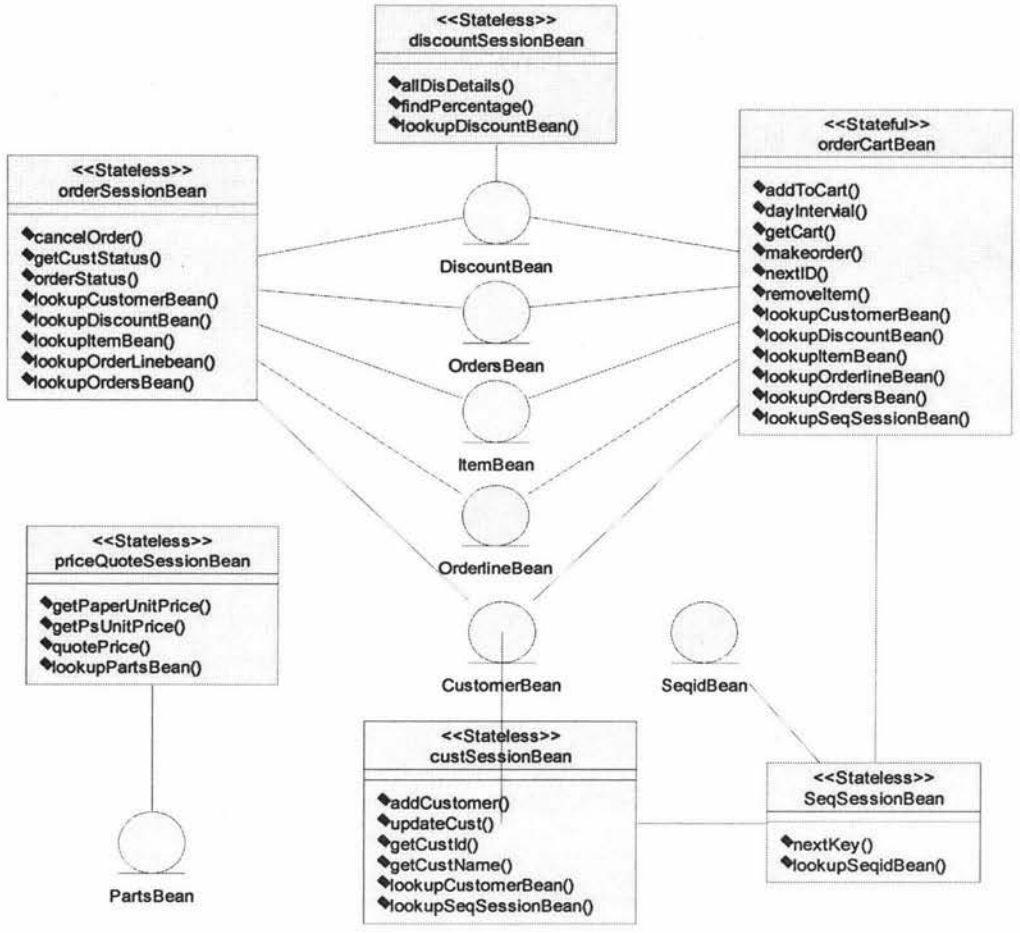


Figure 23: Session Beans Design and the Referenced Entity Beans

As with entity bean implementation, deploying session beans to JBoss also requires JBoss specific deployment descriptors to record the JNDI name of the session bean via XDoclet, but it is simpler than for entity beans because there is no bean fields and relationship in the session bean i.e. *JBosscmp-jdbc.xml* configuration is not applicable to session beans. Only *jboss.xml* document is required.

4.3.3 Web Component Design

Both Servlet and JSP are web components with similar technology use to generate dynamic HTTP web pages. Web components design actually is user interface design. A servlet is Java code with some static HTML that can be load dynamically into and run by the web container. It interacts with clients via a request-response model based on HTTP protocol. JSP is an extension of the servlet technology that allows static HTML with some Java code, which enclose Java code in <% ... %> tags. In general, JSP are better suited for the web pages that contain a large amount of presentation logic and

Servlets are better to perform processing or business logic. In fact, JSP was built on the servlet foundation and needs the servlet technology to work [5]. So in the J2EE application server, the JSP is compiled into a servlet first and then is executed by the web container.

4.3.3.1 Servlets Design

The life cycle of a servlet starts when the web container receives a request the first time. The servlet processes the request and returns the response to the container. And then the container sends the response to the client. Usually the container does not unload the servlet and it remains in memory to serve the next requests [5]. The current version Servlet API provides two packages for Servlet programming. The package *javax.servlet* contains basic classes and interfaces. And another package *javax.servlet.http* extends from. *javax.servlet* and offers more advanced features to make programming easier. Every servlet should implement *javax.servlet.Servlet* interface. In the package *javax.servlet.http*, the most useful class *javax.servlet.http.HttpServlet* represents a servlet that extending from *javax.servlet.GenericServlet,* which contains many useful methods including the six *doXxx* methods e.g. *doPost, doGet, doPut* etc that relate to processing HTTP 1.1 request methods. A *processRequest* method is often used since it can be called by both Post and Get methods. The life cycle of a servlet is determined via the following three methods [5]:

- *init():* it is servlet instantiation method as constructor of a class. The container calls it only once for each servlet.
- *service():* processing request again and again until destroyed, it accepts two parameters *ServletRequest* and *ServletResponse* to communicate with the web container, i.e. get requests from the container and return responses to the container again. The equivalent two objects *HttpServletRequest* and *HttpServletResponse* inside *javax.servlet.http* package are the most used in programming. The syntax shows as follow:

```
protected void service (HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException
```

The information gets in and out of the servlet via calling the properties of objects *request* and *response,* such as:

```
Enumeration names = request.getServerNames();
PrintWriter out = response.getWriter();
```

- *destroy():* called when the web server is shut down or short of memory.

Since the HTTP protocol is stateless, a servlet represents a stateless session too. The solution introduces a session management to keep users' information in several ways: *HttpSession, Hidden fields, Cookies* and *URL rewriting*. The MGProject application uses *HttpSession* to keep track of a client. All servlets in the same application can access the *HttpSession* object. The user information is stored in the *HttpSession* object to be shared among servlets. Example codes for a servlet design and session management from the MGProject application is shown in figure 24.

Since Servlet 2.3, servlet filters were introduced and enables interception and modification of the information in the objects *HttpServletRequest* and *HttpServletResponse* before they are passed to a servlet. This makes servlets design more flexible. The characteristic of a servlet filter was presented in section 2.1.2.1.

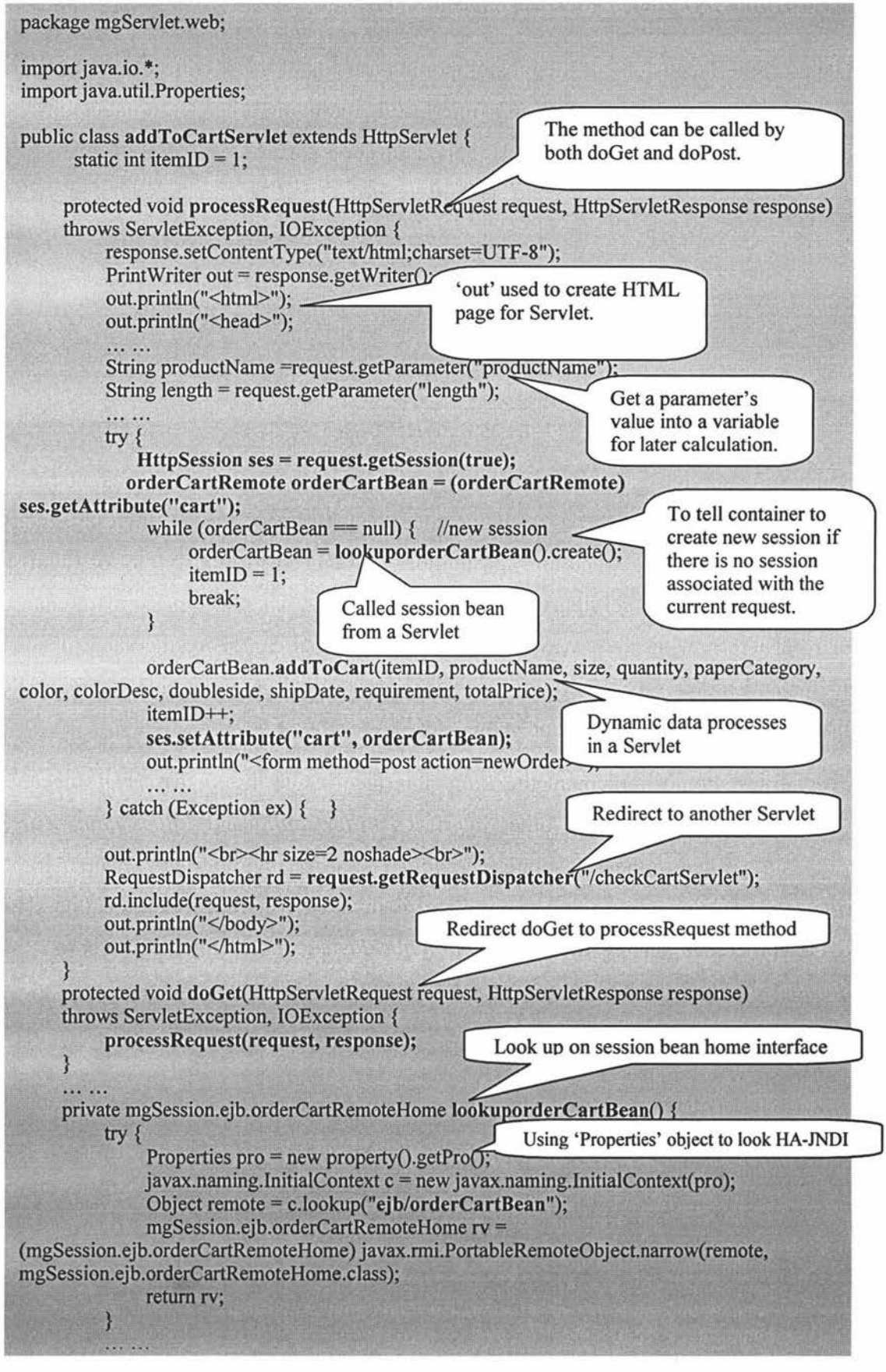


Figure 24: Example Codes for a Servlet Design and Session Management

4.3.3.2 User Interface Design - Servlets Implementation

The user interface is one of the most important aspects of the MGProject application. The common way to design a web user interface is including a header, a navigator and footer. It is the case for this benchmark application see figure 25.

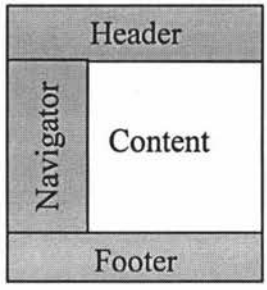


Figure 25: Interface Design

Some or all servlets have the same header, navigator and footer is the most common design style for current web pages. If we code them in every servlet, there are many redundant codes in the application. In addition, if we need to modify any of them, it requires correcting to all servlets that have the same parts. This becomes a maintenance problem. Therefore, the best way is all or some servlets can share the same header, navigator and footer so only coding once. A Servlet filter provides this capability to enable all servlets to share the same contents.

Every filter should implement the *Filter* interface inside the *javax.servlet* package. The package also include two other interfaces *FilterConfig* and *FilterChain*. Three methods are defined in the *Filter* interface: *init(), doFilter()* and *destroy()*. The method *doFilter()* is the filter performing method that is invoked when a servlet is called. The header is processed on the method the *doBeforeProcessing()* and the footer on the *doAfterProcessing(),* the navigator is processeed on the *doFilter*() method.

Header Coding

```
private void doBeforeProcessing(RequestWrapper request, ResponseWrapper response)
    throws IOException, ServletException {
        if (debug) log("headerFooterFilter:DoBeforeProcessing");
        //loading header image
        response.setContentType("image/jpeg");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<center>");
        out.println("<img src=\"images\\mg.jpg\" border=0 ismap height=33% width=75% />");
        out.println("</body>");
        out.println("</html>");
    }
```

Here place an image on the header

Navigator Coding

```
public void doFilter(ServletRequest request, ServletResponse response,
                FilterChain chain) throws IOException, ServletException {
    RequestWrapper  wrappedRequest  = new RequestWrapper((HttpServletRequest)request);
    ResponseWrapper wrappedResponse = new

ResponseWrapper((HttpServletResponse)response);
          doBeforeProcessing(wrappedRequest, wrappedResponse);
          Throwable problem = null;
          try {
                PrintWriter out = response.getWriter();
                out.println("<html>");
                out.println("<body>");
                out.println("<br><br><a href=customerHome><b><font
color=#7a0202>Customer Home</font></b></a><br><br>
                ... ...
                out.println("</body>");
                out.println("</html>");
          } catch(Throwable t) {
                problem = t;
                t.printStackTrace();
          }
          doAfterProcessing(wrappedRequest, wrappedResponse);
          if (problem != null) {
                if (problem instanceof ServletException) throw (ServletException)problem;
                if (problem instanceof IOException) throw (IOException)problem;
                sendProcessingError(problem, response);
          }
    }
```

Design navigator here as designing a HTML page

Footer Coding

```
private void doAfterProcessing(RequestWrapper request, ResponseWrapper response)
      throws IOException, ServletException {
          if (debug) log("headerFooterFilter:DoAfterProcessing");
          PrintWriter out = response.getWriter();
          out.println("<html>");
          out.println("<body>");
          ... ...
          out.println("<p><font color=#10127 size=-1>Web Site Designed and Maintained by
M&G Ltd.,     " + now + "</font></p>");
          ... ...
```

Design footer here as a HTML page

The filter is applied to all servlets in the MGProject application. So the filter will be called before a servlet is processed that ensure each servlet has the same header, navigator and footer. As stated before, each servlet can apply one or more filters. This flexible mechanism allows some servlets to have one content and others have other contents. But for this benchmark application, one filter is used by all servlets.

To fit the system requirement, one servlet filter and nineteen servlets are designed. Some servlets have only static content and some servlets use redirection methods to pass information among servlets. The main advantage of a servlet is that it can process dynamic data. In this case, the processing is via invoking the declared methods on the

remote interface of session beans. Here a brief layout of all servlets and their referenced session beans are given.

- headFooterFilter – Servlet filter is applied to all servlets, it defines the header, navigator and footer of web page. So inside every servlet in this application, we can navigate to other servlets including *customerHome, neworder, orderStatus, cancelOrder, customerStatus* and *custInfoServlet.*
- customerHome – this is the home page of MGProject shown in figure 26.

Figure 26: The Home Page of MGProject

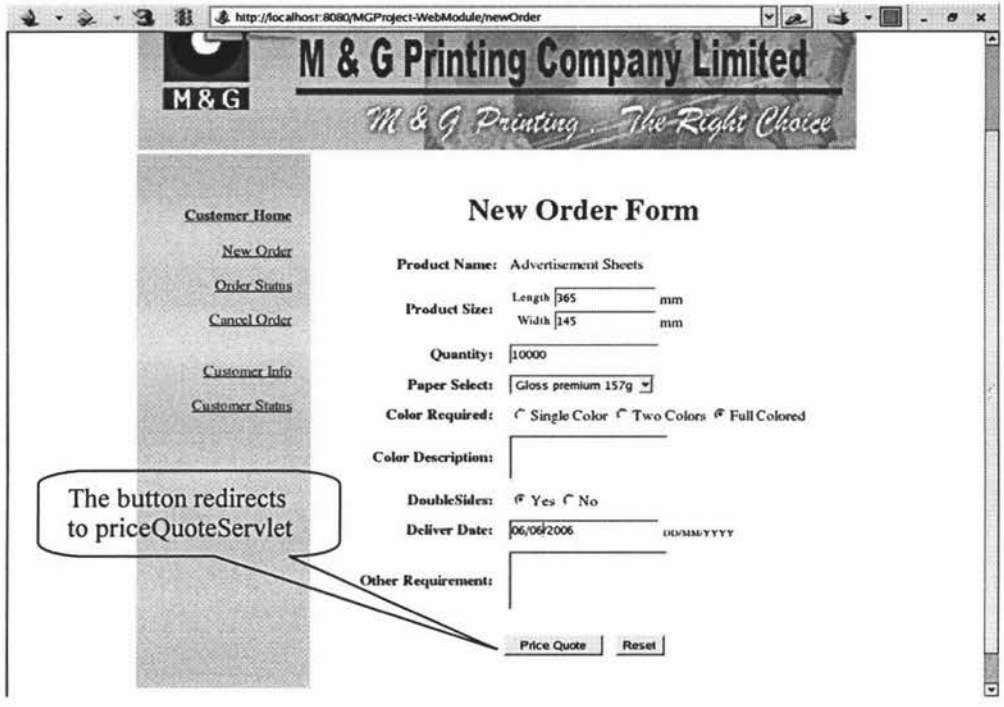


Figure 27: New Order Servlet

- neworder – presents an order form to fill in. This servlet has input validation checking. The priceQuoteServlet servlet will display after submitting the page.

  - priceQuoteServlet – the servlet processes dynamic data invoking the session bean priceQuoteSessionBean. The web page displays the quoted price. The servlet will redirect to addToCartServlet if a customer is satisfied with the price and puts the item in a shopping cart.

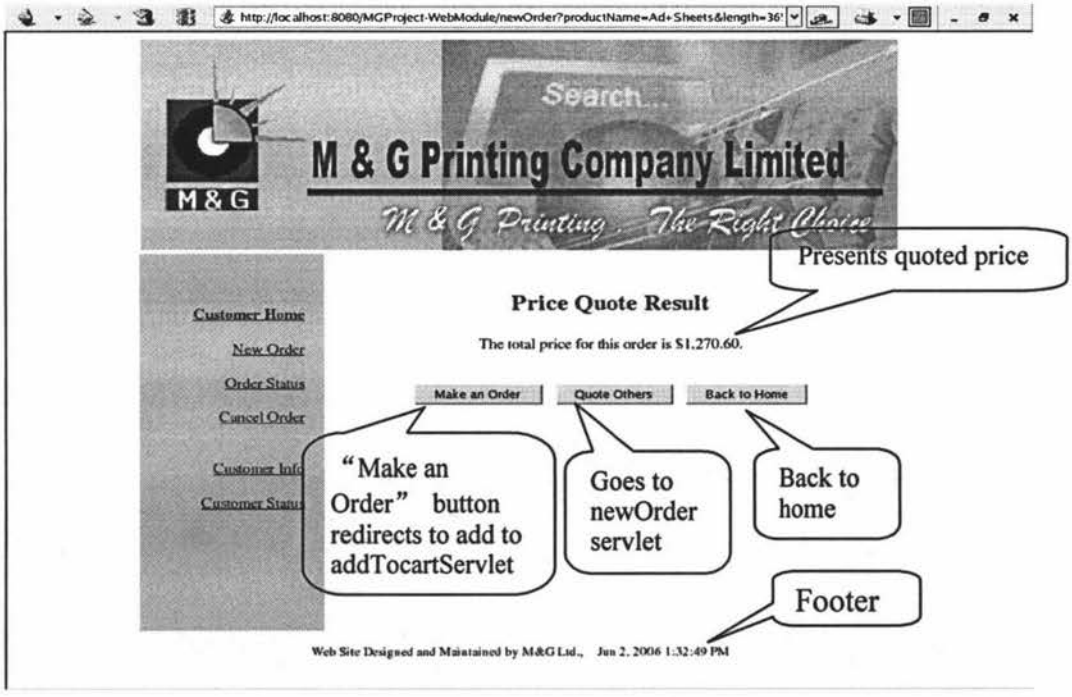


Figure 28: The Interface of priceQuote Servlet

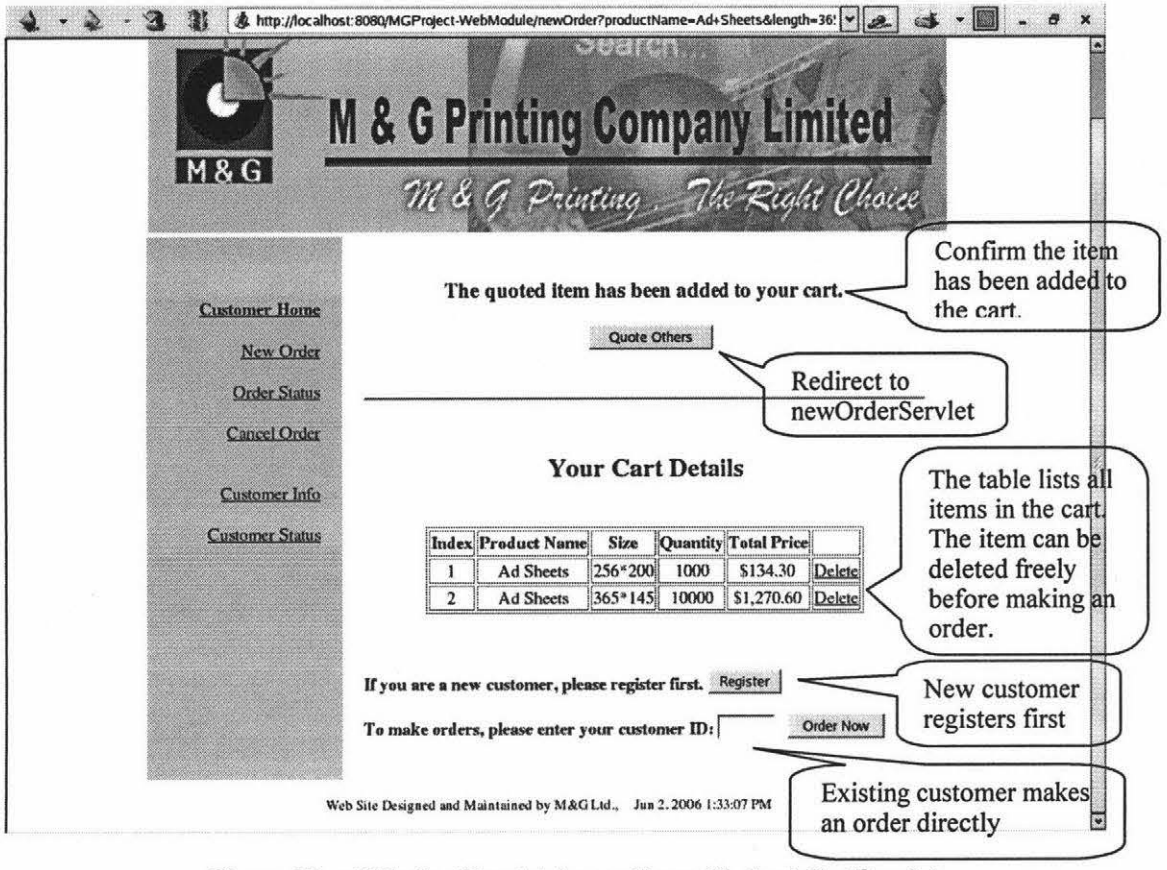


Figure 29: addToCartServlet Appending with checkCartServlet

- addToCartServlet – the servlet will manage a HTTP session to keep the customer shopping state via invoking the stateful session bean orderCartBean. From the servlet, it can be automatically append another servlet --- checkCartServlet and is shown in figure 29.
    - checkCartServlet – the servlet will show all items in the shopping cart. Can cancel the items in the cart. If a new customer, needs to register and it will redirect to registerServlet. Or can directly make an order using the existing customer ID and the orderSuccess servlet will show.
        - ✓ registerServlet – it is a new customer registration form. If the required fields i.e. contact name and phone number are filled in, the page will be redirected to the servlet registerSuccessServlet.

- registerSuccessServlet – the servlet will invoke custSessionBean to add the new customer to the database and show the customer ID (sees figure 30). And it also looks up the session bean discountSessionBean to show the discount percentage for first time customer. From the servlet, you can go back to checkCartServlet or view the discount rules to call discountServlet.

  ⇒ discountServlet – the servlet calls discountSessionBean to get the discount rules from the database. It also can go back to checkCartServlet.

✓ orderSuccess – after a customer makes an order, the servlet will give an order ID to the customer as getting the customer ID in figure 30. The customer uses this order ID and customer ID to check the order status later.

➢ orderStatus – the servlet firstly calls on orderInterface, which is designed for input custID and orderID when a customer is checking an order status or cancels an order. After inputting valid number, the orderList servlet will display.

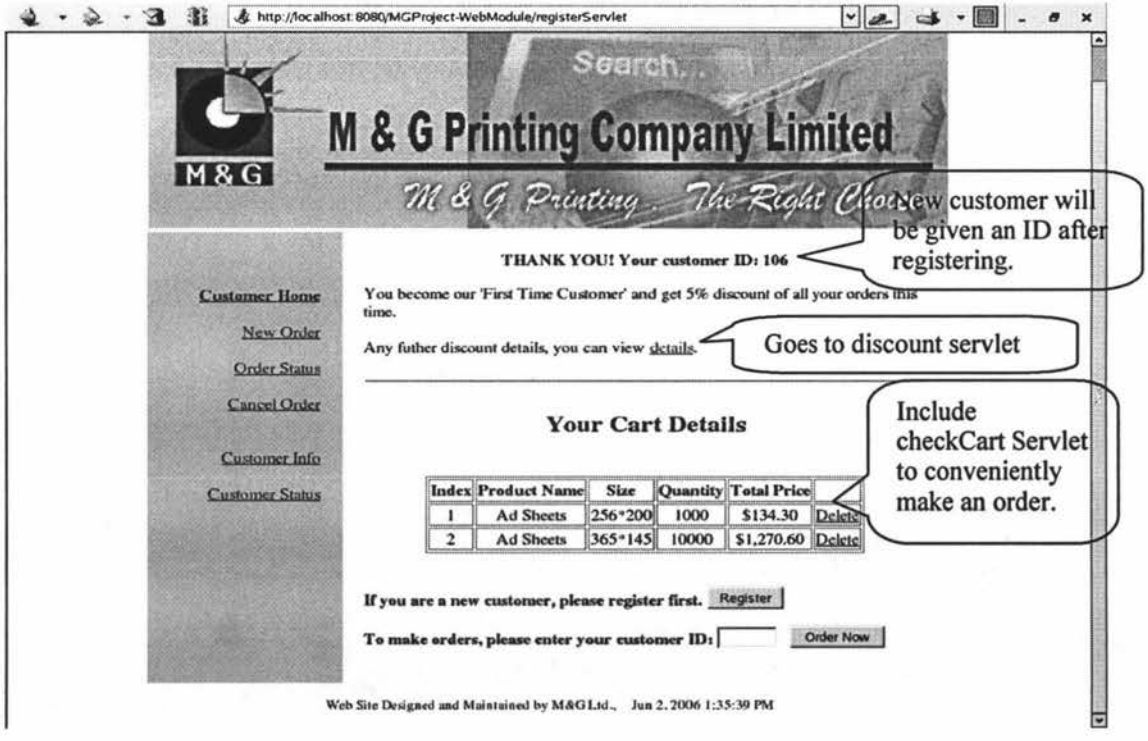


Figure 30: The Interface of registerSuccessServlet

- ❖ orderList – the servlet invokes the orderSessionBean to check if custID and orderID match to values store in the database. If it is so, then the detail of order information will be displayed (sees figure 31). Otherwise the orderInterface displays to require input again.

➢ cancelOrder – the servlet calls on orderInterface like orderStatus. After entering custID and orderID, the cancelSelect servlet displays (sees figure 32).

- ❖ cancelSelect – the servlet will invoke orderSesisonBean as orderList to show all orderline item in the order. The customer can perform the cancel action to delete all or part of orderline items if they are not in a "processing"status. The cancelConfirm servlet displays after deleting an item.

Figure 31: The Interface of orderList

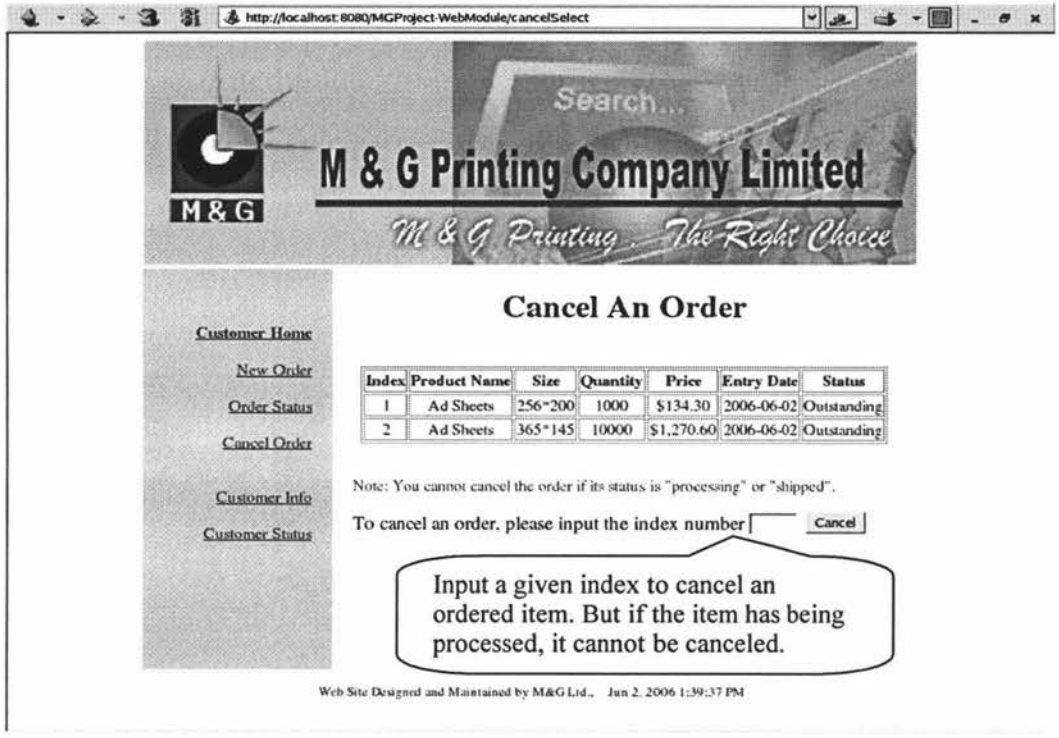


Figure 32: The Interface of cancelSelect Servlet

- cancelConfirm – the servlet invokes orderSessionBean to update the data record in the database.

➢ customerStatus – the servlet requires the customer ID and then invokes the custSessionBean and orderSessionBean to list out all outstanding orders related to the customer (sees figure 33).

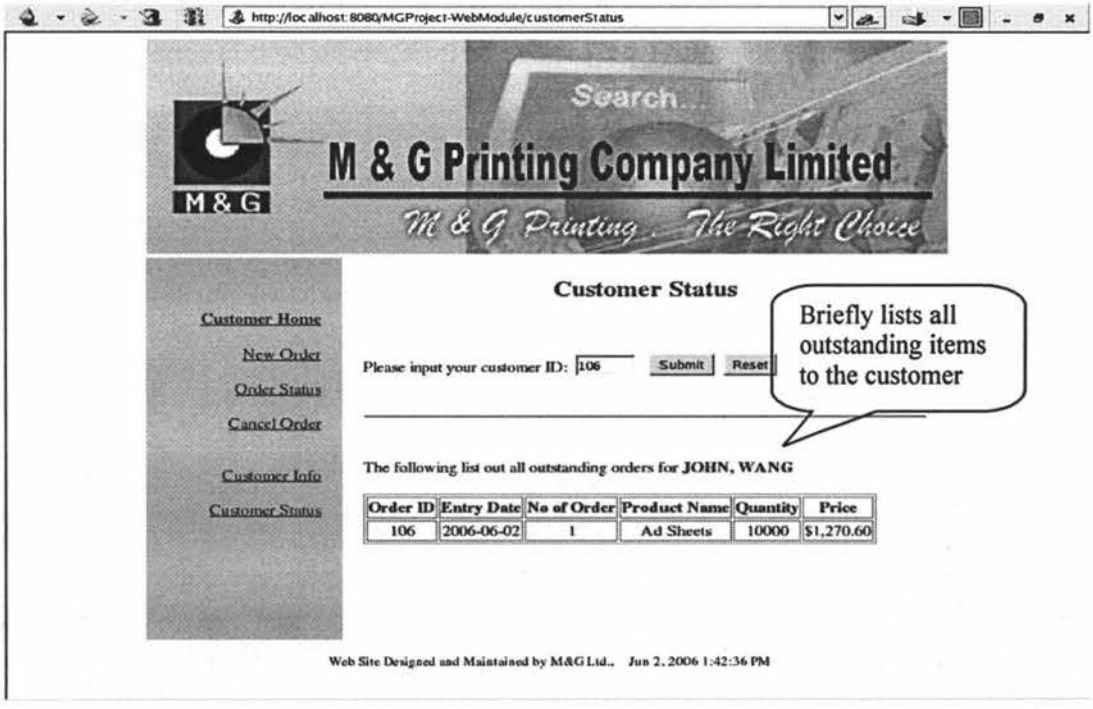


Figure 33: Display the Status of Outstanding Orders for the Customer

- custInfoServlet – the servlet will redirect to custID servlet if they forget their custID. After giving the customer ID, the customer personal information will be display on custDetail servlet.

    - custID – the servlet requires the customer name and contact phone number to find custID via invoking the custSessionBean.
    - custDetail – the servlet calls on custSessionBean to display the personal registration information. The customer can update the information and save to the database.

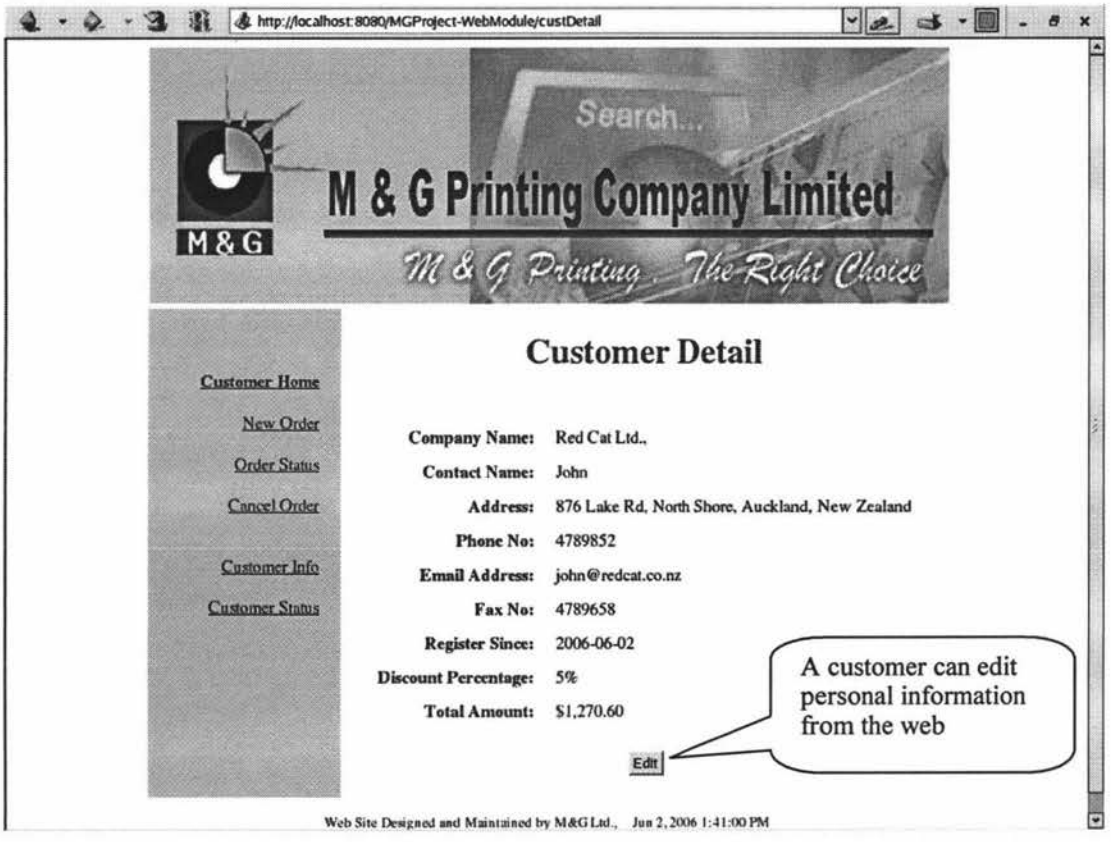


Figure 34: The Interface of custDetail Servlet

Chapter 5: Load-Balancing Policies Design

Summary: This chapter covers the detailed design of the dynamic and dynamic weight-based load-balancing policies for clustered JBoss. The policies will extend JBoss LoadBalancePolicy interface and utilizes EJB timer service and JBoss TreeCache. The LoadBalancePolicy interface exposes the cluster members' information. The EJB timer service performs tasks in a regular period and JBoss TreeCache provides shared storage for all timer session beans located in different machines. In addition, the chapter also lists the pseudo code design.

After completing a benchmark J2EE application design and implementation, another challenging task is to design load-balancing policies that are applied to a clustered environment. The application will run on the JBoss application server, which has four kinds of load-balancing policies built-in. However, the built-in load-balancing policies became the bottleneck of the scalability performance test since the built in load-balancing policies are static. Dynamic and dynamic weight-based load-balancing policies, which will extend JBoss LoadBalancePolicy interface are designed and discussed in this section. From the design concept, the dynamic load-balancing algorithm is more sophisticated. This is expected improve the performance since load will be adjusted based on the current state of the machine.

5.1 Design Construction – using EJB Timer Service & JBoss TreeCache

The dynamic and dynamic weight-based load-balancing policies will detect every server's workload before dispatching the requests to a server. The current architecture is based on a Beowulf clustered system Sisters – a Linux based system provides system load information in the file */proc/loadavg*. The information indicates the average number of processes waiting to run on a CPU at the last 1, 5 and 15 minutes. The load information is updated frequently in the Linux system. A mechanism that can read this updated information in a regular period and automatically refresh the workload record frequently is required.

5.1.1 The EJB Timer Service

As stated in section 2.1.2.1, the EJB timer can auto invoke the *ejbTimeout* method once or many times when the timer expires. Even if the server is shut down or crashes, the timer is still persistent and is saved in the database. In this application, the *ejbTimeout* method can be called again and again in a regular period. Some transactions can be processing after the time out and then the timer starts again. The timer service creates an interval timer in this application as in figure 35:

```
timerService = context.getTimerService();
Timer timer = timerService.createTimer(5000, 5000, null);
```

Figure 35: EJB Timer Service Creates an Interval Timer

Here, the interval timer is invoked every 5000ms that starts after a specified duration 5000ms (refers to the first value 5000 ms). There is no information associated with the timer. After time out, the method *ejbTimeout* is invoked automatically.

```
void ejbTimeout () {
        read_in _updated_sys_load_info;
        refresh_data;
        ...
}
```

Figure 36: The Transactions Processed after Time Out

The transactions inside ejbTimeout method will only be processed when the time out event occurs. The EJB timer service can be applied to all EJBs except stateful session beans. A timer session bean is used in this application to implement the EJB timer service.

In a cluster the timer session bean only can get the load information for the machine it is running on. The solution is to use the JBoss Cache to successfully store shared data.

5.1.2 The JBoss TreeCache

"In-memory caching is a crucial feature in today's large-scale enterprise applications, where scalability and high performance are required. [49]" So using JBoss cache should be the best location to store the load information since:

The information stored in the cache is processed faster than in a database or hard disk.

Usually the cache size is a few hundreds KB and the size of load information is small.

The JBoss cache can be replicated across the cluster that has been built in the JBoss as an MBean (abbreviation of Managed Bean, it is a Java object that exposes a standard

interface to access and control resources) service of application server.

The JBoss server supports TreeCache and TreeCacheAop. Both of them can be replicated synchronously or asynchronously. The TreeCacheAop extends the functionality of TreeCache to handle more complex transactions and objects [50]. The TreeCache is used in this application performing a simple replicated storage function.

JBoss cache can run standalone or integrated with JBoss service as an MBean service. It supports three different cache modes: LOCAL, REPL_SYNC and REPL_ASYNC. The LOCAL mode is only accessed for one machine. REPL_ASYNC mode can support non-blocking replication that is useful in heavy load environment, which is needed in this application.

Although JBoss *TreeCache* can run as a MBean service inside the JBoss, it cannot be used in the program without extra configuration. First, the *jboss-cache.jar* file has to be copied to the lib directory and JBoss has to be restarted. Then a regular XML-based *TreeCache* configuration file has to be copied to the */deploy* directory [50]. But the *TreeCache* configuration file is not found in JBoss 4.0.3. The XML file is available via download from http://www.jboss.org/products/jbosscache/downloads. The sharing version is used in this application. After extra configuration, the *TreeCache* service is used via calling the MBean service as shown in figure 37:

```
MBeanServer server=MBeanServerLocator.locateJBoss();
TreeCacheMBean cache;
cache=(TreeCacheMBean)MBeanProxyExt.create(TreeCacheMBean.class,
        "jboss.cache:service=TreeCache", server);
cache.put("/a/b/c", null);
```

Figure 37: Using TreeCache as JBoss MBean Service

Here, "/a/b/c" represent the third level node "/c" in the cache tree. Each node has only one parent, "/a" is the root node. And "/b" has a parent node "/a" and a child node "/c". Each node actually is a HashMap. A HashMap object (key& value pair) can be put in the cache and retrieved later. In this application, the pair refers to host IP address and load values in that machine. The cache is replicated and shared among the clustered JBoss. So the load information of all cluster machines can be obtained easily for each timer session bean.

5.2 Dynamic Load-Balancing Policy Design

The design concept for the dynamic load-balancing algorithm is that it always finds a

node with the lowest load values i.e. the idlest node. Assume each clustered node runs on a different Linux machine. That means each machine node has a unique IP address and load average information. This pair of information would be stored in the JBoss TreeCache for comparison reasons. Figure 38 shows an example where there are three nodes in the same clustered partition.

Node	Load Average
192.168.1.253 (sisters)	2.91, 2.33, 2.10
192.168.1.101 (amd1)	1.04, 1.01, 1.00
192.168.1.102 (amd2)	0.00, 0.00, 0.00

Figure 38: Pair of Information in JBoss TreeCache

The load average indicates the average number of processes waiting to run on a CPU at the last 1, 5 and 15 minutes. Although the average value cannot exactly reflect the CPU current usage, it reflects the workloads of the nodes in a cluster. The comparison starts from the most important value - the last one-minute number of processes in the queue. If all nodes have the same value, the second load value will be compared. The third value needs to compare only if the first two values are the same. After comparing the above values, we know 192.168.1.102 is the idlest node in the cluster. So the node 192.168.1.102 will be selected to serve all client requests until a new load average value is read in.

5.2.1 The JBoss *LoadBalancePolicy* Interface

The JBoss *LoadBalancePolicy* interface is the basic interface for designing load-balancing polices. There are four policies implementing classes including *FirstAvailable, FirstAvailableIdenticalAllProxies, RandomRobin,* and *RoundRobin.* The explanation of these four built-in policies is given in section 3.2.4.2. The interface defines one variable *serialVersionUID* and three methods. The most important method is *chooseTarget* that is an overloaded method in the interface:

```
public Object chooseTarget (FamilyClusterInfo clusterFamily,
                            org.jboss.invocation.Invocation routingDecision)
```

Figure 39: Overloaded Method chooseTarget in JBoss *LoadBalancePolicy* Interface

Here, the passed parameter *clusterFamily* contains a list of potential target nodes and *routingDecision* that refers to the actual invocation object. The *clusterFamily* is the most important parameter that contains all available cluster members' information that will be downloaded by clients. The mechanism allows the clients to choose a target

node from the available list based on the load-balancing policy. The following shows what information is downloaded to client stub.

```
target[0]=org.jboss.invocation.jrmp.server.JRMPInvoker_Stub[RemoteStub [ref:
[endpoint:[192.168.1.64:4447](remote),objID:[1749757:108e561090b:-7fff, 3]]]]
```

The client can get host IP from the Info

Figure 40: Downloaded Information in Client Stub

After parsing the above information, we can get the host IP address. Another way to get server host name is via object *JRMPInvoker_Stub*. Therefore, usually a prefered node is selected based on a load-balancing policy, if the IP address of a target node matches the selected node by the load-balancing algorithm, then this target node will be returned to serve a client. For the dynamic load-balancing policy, the dynamic policy always selects the idlest node as the preferred node to serve a client.

5.2.2 Pseudo Coding for Dynamic Load-Balancing Policy

Before processing the dynamic policy, there are many actions processed by the session bean *tmSessionBean*, which implements the EJB timer service to schedule all the basic transactions for dynamic and dynamic weight-based policy. The jobs done by this tmSessionBean are shown in the following pseudo code in figure 41 and the figure also shows how the dynamic load-balancing policy invokes with tmSessionBean.

```
public class dynamicCache implements org.jboss.ha.framework.interfaces.LoadBalancePolicy {

    public dynamicCache() {    }

   public void init (HARMIClient father) { }

    public Object chooseTarget (FamilyClusterInfo clusterFamily) {
        return chooseTarget (clusterFamily, null);
    }

   public Object chooseTarget (FamilyClusterInfo clusterFamily, Invocation routingDecision) {
      targets = clusterFamily.getTargets();
      if targets_size=0      return null;
      if targets_size=1      return targets(0);
      parse targets information and store all clustered IP to HashMap (IP, targetNode);
      preferNode = tmSessionBean.bestNode;
      while (!HashMap(preferNode) {
            remove bestNode from local HashMap of tmSessionBean //server may fail
            preferNode = find new bestNode;
      }
      return HashMap(bestNode) i.e targetNode;
   }
}
```

```
class tmSessionBean {
        ejbCreate() {
                getHostName;
                read-in machine average load value;
                load average load value into TreeCache;
                get all clustered machines' load values from TreeCache and store in local HashMap;
                do comparison of load values for all clustered nodes;
                bestNode = idlest-node;
        }
        ejbTimeout() {
                read-in machine average load value;
                if load value changed {
                        load average load value into TreeCache;
                }
                get all clustered machines' load values from TreeCache and refresh local HashMap;
                do comparison of load values for all clustered nodes;
                bestNode = idlest-node;
                keep one timer for all tmSessionBean and cancel others
        }
        ......
}
```

Figure 41: Pseudo Code for Dynamic Load-Balancing Policy

5.3 Dynamic Weight-Based Load-Balancing Policy Design

The above selection based on the dynamic load-balancing policy always chooses the idlest node in the cluster. But with the load average updated about every minute, the interval is long enough to turn the idlest node into the busiest node when the cluster faces heavy workloads. A weight-based mechanism would use the first load average value (the most important value i.e. 2.91, 1.04, 0.00 in the case). The mechanism is introduced to share workloads within all nodes in the cluster. The solution is to calculate the common multiple for these values. This value is used as the selection ratio. Here, there has one value 0.00, which should lead to the result of common multiple be zero. So firstly the value 0.00 has to be modified to a meaningful value that is larger than zero but less than other load values. Since the precision of load values to hundredth, the meaningful value may extend to thousandth like 0.001. To ensure the each node at least runs once, the selection ratio for the node with the least common multiple will set to 1, for example

Node	Load Average	Common Multiple	Selection Ratio
192.168.1.253	2.91	1.04*0.001 = 0.00104	0.00104/0.00104 = 1
192.168.1.101	1.04	2.91*0.001 = 0.00294	0.00298/0.00104 ≈ 3
192.168.1.102	0.001	2.91*1.04 = 3.0264	3.02640/0.00104 = 2910

Figure 42: Calculation the Ration for the Weight-Based Policy

Since the 192.168.1.102 is fully idle, the weight-based solution should be similar to dynamic solution. But it should improve performance when these nodes have similar load average. Assume the load average for 192.168.1.102 is 1.00, and then the selection ratio would be 1, 3, 3.

5.3.1 Implementation for Dynamic Weight-based Load-Balancing Policy

The design steps of the weight-based load-balancing policy are very similar to the dynamic policy. Most actions are common for both policies such as *getHostName*, *getLoadAverage*, loading to *TreeCache* and store in local *HashMap* etc. A serializable object *cRatio* that can be shared among different EJB is created.

```
public class cRatio implements java.io.Serializable {

    /** Creates a new instance of cRatio */
    public int count;   //count selection ratio
    public String host;   //hostname
    public HashMap countRatioMap; //EJB local object
}
```

Figure 43: Design a serializable object cRatio

In the session bean *tmSessionBean* class, a method *refreshRation()* calculates the ratio in use. The ratio is stored in a local *HashMap* object as a pair of hostname and ratio. This *HashMap* object contains all clustered hostname and its ratio values. The idlest host is put on the top of the *HashMap*. So every time when a client has a request, the policy will select a host from *HashMap* list and the ratio count number will reduce by 1 (this value is shared among different session bean via object *cRatio*). If the ratio count value becomes 0, the related hostname will be removed from the *HashMap* list. And the next host will be selected until its ratio count value turns to 0. The same processing will be repeated until nothing is left in the *HashMap*. Then the *HashMap* is reloaded with new calculated values. The weight-based load-balancing policy is dynamic since the calculation of ratio value is changing based on EJB timer schedule.

The above solution is a more accurate way to allocate the clients tasks among clustered application servers. It basically ensures every server has equal load according to the current status of the server.

Chapter 6: Test Plan

Summary: The test plan in this chapter has two meanings. One refers to establishing a test plan in Jmeter. The plan will simulate a large number of independent clients performing online actions to interact with back-end servers. Another means to plan a test procedure for testing the performance of JBoss under various load-balancing policies. This plan should provide a sequence of testing steps to achieve the testing goal. Meanwhile, techniques to ensure the benchmark application runs as smoothly as possible under the very heavy loads are discussed.

6.1 Test Plan in Jmeter

A benefit of Jmeter is that it is not only a testing framework but also a sophisticated GUI for constructing tests. A test unit in Jmeter is called a test plan. Some basic steps to construct a test plan were presented in section 3.2.5. Here, details of the test plan for this study are presented and the interface of Jmeter is shown in figure 44.

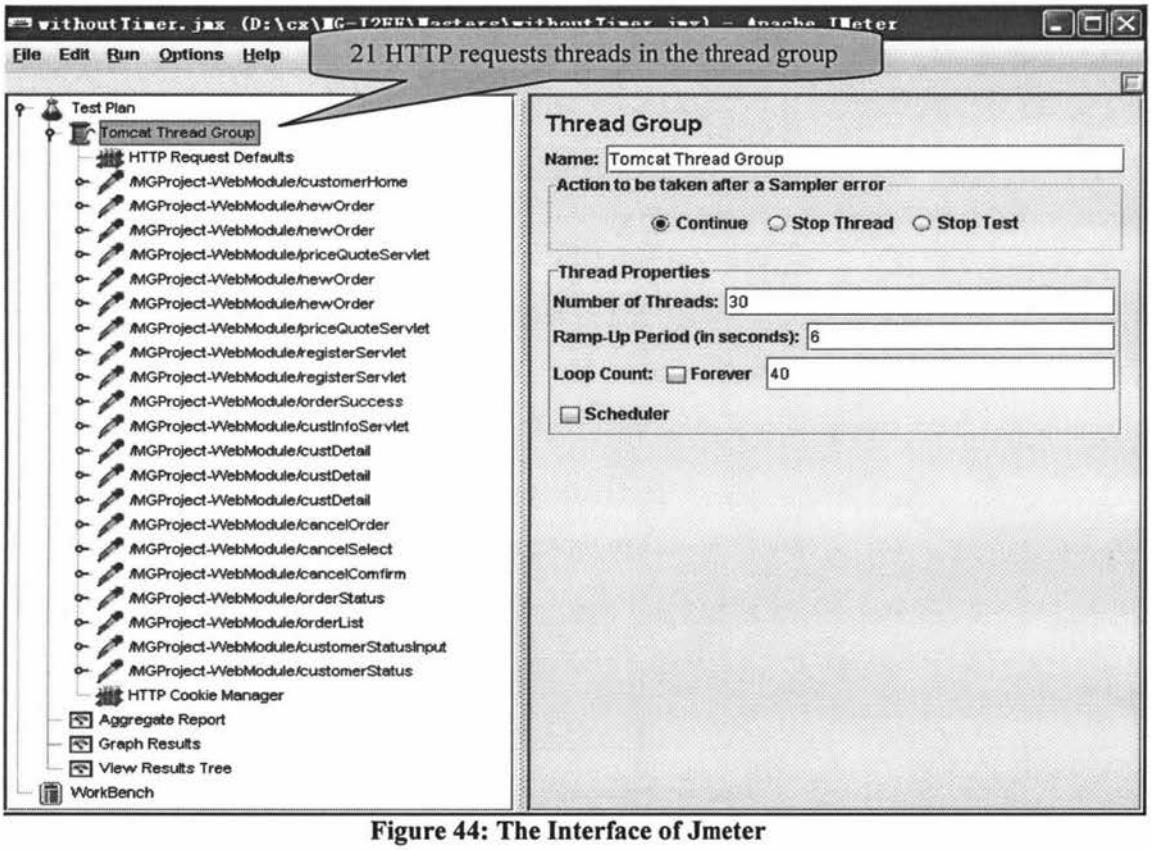


Figure 44: The Interface of Jmeter

The test plan begins by adding a thread group. A thread group contains a set of threads and each of them is independent of each other. Each thread can make HTTP requests. The configuration of a HTTP request has to specify the name or IP address of web server, listening port number, protocol and path for HTTP request. This would be a very tedious task if all HTTP requests were sent to the same web server. Jmeter provides a HTTP Request Defaults Control Panel, which applies to all HTTP request in the thread group. Each HTTP request represents a servlet in the MGProject application. In order to simulate online client behaviors, Jmeter acts as a HTTP Proxy Server to record the sequence of actions on the servlets of MGProject, including static and dynamic data. A timer to each HTTP request to simulate thinking time e.g. viewing information, filling form and making decision etc can be added. For detailed configuration see Jmeter Tips [51].

There are a total of 21 HTTP requests threads created via the HTTP Proxy Server for this test plan. Each thread is independent, then the 21 requests may represent 21 clients' behaviour in Jmeter. These requests include the main servlets created for the application such as new customer registration, making order, canceling order and checking customer information etc. Most of them have dynamic data processing. Jmeter can record all parameter and value pairs in the field of *Send Parameters With the Request* in HTTP configuration page, as if a real client inputs values to fill a form. For example, while checking the status of an order, the servlet *orderStatus* displays and requires two fields that have to be completed before sending the request, i.e. providing customer ID ${*custID*} and order ID ${*orderID*}. This is configured in the *Send Parameters With the Request* field in Jmeter. Some parameter and value pairs in a servlet are only available after processing another servlet. For instance, a new customer has to give personal information e.g. name, phone number to register first and then get a customer ID ${*custID*} number from the previous servlet. Such information is gained in Jmeter via a Post-Processor Element and *__regexFunction.*

The regex function uses any regular expression to parse the previous response. The detail on the use of *__regexFunction* is in the Jmeter Manual [11]. An example from the test plan configuration using *__regexFunction* is given in figure 45.

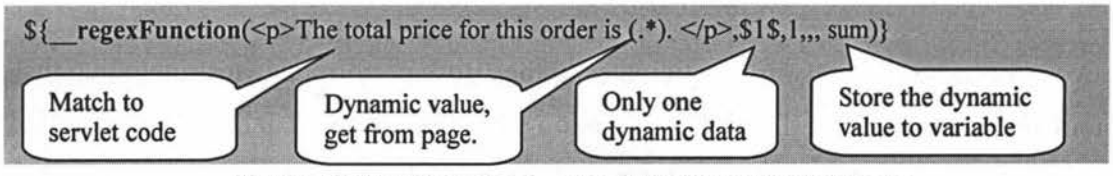


Figure 45: An Example of __regexFunction using in Jmeter

The test plan actually presents a single client's behaviour. For performing a stress test, the test can be more realistic since variable data participates in the test. The servlet *registrationServlet* records new customer's personal information and *priceQuote* represents the quote price for various printing materials e.g. size, color requirement etc. Through the Pre-Processor Element and adding user parameters before these two HTTP requests, ten users parameters are configured to participate in the test. But customer ID ${*custID*} and order ID ${*orderID*} values are retrieved dynamically from the database via *__regexFunction*. The online behaviors of each of the ten clients are similar. The test plan simulates various clients interacting with the web server.

Since each HTTP request represents a stateless session, the shopping cart information cannot pass among different HTTP requests. A mechanism to maintain session information for the application is provided via the Jmeter cookie support for web application. The servlets use a HTTP session object to manage session data, so Jmeter uses the HTTP Cookie Manager to keep session state.

Two versions of the test plan were created for later use, with thinking time and without thinking time. The difference between them is that the thinking time version adds a random time (ms) for each HTTP requests in the test plan. The thinking time models more realistic behaviour associated with a client.

To configure the test plan:

1. Create a thread group
2. Add HTTP Request Defaults Control Panel to the thread group
3. Configure Jmeter as a HTTP Proxy Server to automatically generate HTTP requests
4. Use *__regexFunction* to parse the previous response and fetch needed information
5. Simulate different clients to add various users' parameters in the HTTP request
6. Add HTTP Cookie Manager to maintain session state.

6.2 Stress Test Plan

Several open source software participates in the JBoss performance test including the test tool Jmeter, web server Tomcat, application server JBoss and database PostgreSQL. The default parameters settings in these software cannot support a stress test. The

bottleneck for the stress test should occur on JBoss and not on other software. Since the final tests focus on comparing the various load-balancing policies affecting the performance of clustered JBoss not for the performance of other applications.

6.2.1 Plan for Jmeter

The interface of Jmeter is shown on figure 44. From the figure, it is easy to know the Jmeter performs stress tests is based-on three parameters:

- *Number of threads*
- *Ramp-up period*
- *Loop count*

} See section 3.2.5 for detailed explanations

As the explanations in section 3.2.5, these three parameters should affect the performance of a server. Increasing the number of threads indicates increasing the number of clients. And shortening the ramp-up period means a server has to handle more requests in a time unit. And the bigger loops value makes the server process requests for longer. The experiment cannot compare and analyze all these parameters at the same time. So two parameters have to be fixed. Considering many experiments are required and each experiment should not run too long, so loops value should be fixed in a suitable value. In addition, if the server handles too many requests in a time unit that may cause many threads waiting for sharing resource and the "TimeOut" exception may be thrown early, then the second parameter ramp-up period also has to be fixed in a suitable value. The last parameter, number of threads can directly simulate the number of clients to interact with the server. That should be the most important parameter to the experiments. On the other hand, the stress test also needs to consider which version of the test plan is more suitable for the test, with thinking time or without thinking time.

In addition, the Java application is very sensitive to JVM heap size. The default heap size in Jmeter is –Xms 256m –Xmx 256m. After some initial testing, the "OutOfMemory" exception is thrown. Particularly if the memory intensive listeners are added in the test plan such as "View Tree Results", they will be memory consumers in the stress test while increasing the simulated clients and loop count. The best listeners to use for long-term stress test are *Aggregate Listener*, *Graph Listener* and *Spline Listener* [46]. Therefore, the "*View Tree Results*", "*Aggregate Listener*" and "*Graph Listener*" are used for recording the stress test results. In Sisters, all nodes have 1G memory. The heap size is increased to –Xms 256m –Xmx 768m so that the memory is not an issue.

6.2.2 Plan for Tomcat

The JBoss is integrated with web server Tomcat 5.5 in the current version 4. When doing the scalability test for the JBoss, usually it is to cluster two or more JBoss. If without configuring, the default load-balancing policy is applied. Each JBoss worked perfectly to handle equal requests under default *RoundRobin policy* for the application client. That means the jobs can be shared among clustered JBoss. However, the requests are handled in a different way for an application client program with JBoss and web application via the web server Tomcat, which resides with JBoss in same JVM. Since the client and server work in different JVM, the client uses RMI protocol to download stub code before sending requests (the scenario sees figure 7 in section 3.2.4.2). But for Tomcat residing in JBoss, there is no need to use RMI protocol i.e. the stub code will never be downloaded. It looks up the local JNDI instead of HA-JNDI without any clustering information (sees figure 46). Therefore, clustered JBoss actually acts as single JBoss for the web application but Tomcat has to do state replication among cluster members.

Separating the Tomcat from the JBoss, means configuring a standalone Tomcat. For configuration of standalone Tomcat sees section 3.2.3. The clustered JBoss works well in this scenario, load balancing requests from Tomcat (sees figure 60 in section 7.2.3). But if Tomcat and JBoss run on the same machine, both of them are critical servers for this application. It will hard to know which server has caused the bottleneck. Therefore, Tomcat and JBoss are run on different machines.

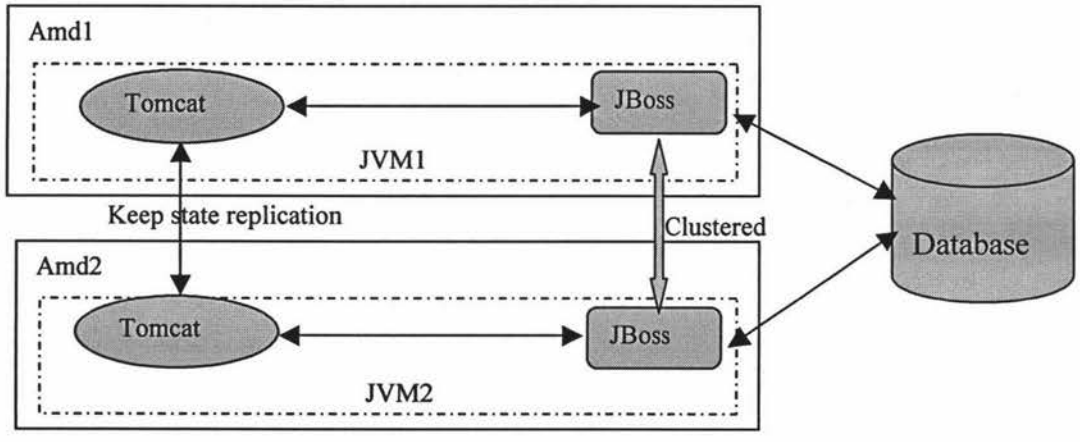


Figure 46: Clustered JBoss with Reside Tomcat for Web Application

However, the processing capability of JBoss increases dramatically after running Tomcat on another machine. Directly increasing workloads to Tomcat and in turn to stress on JBoss, it seems the Tomcat reaches the bottleneck first. Clustering Tomcat or directly starting more standalone Tomcat is a solution to put stress on JBoss.

6.2.3 Plan for JBoss

After separating Tomcat from JBoss, the main function remaining in JBoss is to process business data and logic i.e. managing entity beans and session beans. By default, JBoss uses pessimistic locking to keep entity beans' state consistent. The pessimistic locking is achieved via row-locking on the database table. When an entity bean is in transaction mode i.e. doing reading or writing, an exclusive lock is acquired for the relative row in the table. During locking, any other transactions trying to access the bean's state are blocked until the previous transaction is committed and the lock is released. The pessimistic locking policy guarantees consistency of the beans' state, but causes performance drawbacks too. The policy also blocks the read-only operations. The problem leads to bad performance for J2EE applications. JBoss offers an optimistic policy to handle this situation. In the optimistic policy concurrent access is permitted but only one of the operations can modify the state of the bean [45]. That means concurrently reading the state of bean is not blocked during the transactions but only one modifying operation can perform actions at one time. The implementation of optimistic policy is via defining an entity bean as "read-only" or "read-only" is applied to some methods.

In each entity bean, reading the state of a bean is via calling methods with prefix *get,* e.g *getCustID, getOrderStatus* etc. And modifying the bean's state is via methods with prefix *set*, e.g. *setCustName, setOrderStatus* etc. Therefore, all methods are defined in entity beans with prefix *getXXX* are read-only. The implementation in IDE is via XDoclet definition, such as shown in figure 47.

```
/**
*@jboss.method-attributes   pattern="get*"
*                           read-only="true"
*/
```

Figure 47: Define Read-Only Methods via XDoclet

Such information in the IDE will translate into the *jboss.xml* deployment descriptor as shown in figure 48.

```
<entity>
  <ejb-name>OrdersBean</ejb-name>
    <method-attributes>
      <method>
        <method-name>get*</method-name>
        <read-only>true</read-only>
      </method>
    <method-attributes>
</entity>
```

Figure 48: Read-Only Methods in the jboss.xml

In addition as mentioned in section 3.2.4.3 about commit options in JBoss, the EJB container can be configured to use any Commit Option A, B, C or D. By default, the EJB container uses Commit Option B to manage entity beans. JBoss uses Cache Invalidation Framework to enable Commit Option A for more than one containers to handle entity beans via using cache for faster transactions. So some data may access database only once. As a result, the configuration of the EJB container is modified for entity beans except the *SeqidBean*. For sample EJB container configuration see figure 49.

```
<entity>
  <ejb-name>OrdersBean</ejb-name>
    <configuration-name>Standard CMP 2.x EntityBean with cache invalidation
    </configuration-name>
    ... ...
    <cache-invalidation>True</cache-invalidation>
</entity>
```

Figure 49: Define EJB Container Configuration for EJB

The *SeqidBean* is a special bean created for programming. It is critical for providing the next unique key value for the entity *customer, orders, orderline* and *item*. Such data is not shared so it cannot be kept in cache for reuse. The option C has stronger requirement to keep consistent data in the database than B. Therefore, the Commit Option C is chosen for the container to manage the *SeqidBean* bean.

There is no data sharing to maintain among session beans. However, the study will focus on optimistic load-balancing policies design. The policies will be applied at EJB level to examine the performance of JBoss. The session bean is in the front line of the EJB container. That means the load-balancing policy will be applied to session beans in a cluster environment. Each bean has two interfaces, home and component (bean). So

the load-balancing policies should apply at both home and bean interfaces. The "JBoss AS Clustering" [44] states how to cluster EJB in *jboss.xml* deployment descriptor. Figure 50 presents the configuration for the stateless session beans with load-balancing policy via XDoclet in IDE and figure 51 shows the configuration for the stateful session beans.

To implement the different load-balancing policy for a session bean, the home and bean policy for each session bean is modified via the XDoclet tag in the IDE, which in turn modifies the setting in *jboss.xml*.

For Stateless Session Beans

```
/**
*@ jboss.clustered clustered=true
*@ jboss.cluster-config partition-name="DefaultPartition"
*                       home-policy="org.jboss.ha.framework.interfaces.FirstAvailable"
*                       bean-policy="org.jboss.ha.framework.interfaces.FirstAvailable"
*/                                                                              [44]
```

Home and bean can use different policy for session bean

Figure 50: Clustering Stateless Session Beans with Load-balancing Policy via XDoclet

For Stateful Session Beans

```
/**
*@jboss.clustered clustered=true
*@jboss.cluster-config partition-name="DefaultPartition"
*                       home-policy="org.jboss.ha.framework.interfaces.RoundRobin"
*                       bean-policy="org.jboss.ha.framework.interfaces.FirstAvailable"
*                       state-manager="HASessionState/Default"
*/
```

The bean policy of stateful bean should use FirstAvailable policy to be sticky on a machine to maintain the session state

Add state manager to stateful bean

Figure 51: Clustering Stateful Session Beans with Load-balancing Policy via XDoclet

6.2.4 Plan for PostgreSQL

The database is one of easier points to become bottleneck in an application. Almost all modern databases have their own data concurrently control mechanism for persistent data during transactions. The PostgreSQL provides Multiversion Concurrency Control (MVCC) to adjust the database performance in a multi-user environment by using transaction isolation, read committed isolation level, serializable isolation level and explicit locking like table or row level locking [53]. Read committed isolation level is the default setting in PostgreSQL, which is enable to fetch only committed data.

The default setting also causes problem in the MGProject application for the *SeqidBean* entity. Since the main task for this special bean is to provide a unique number block and store it in a shared object *SequenceBlock* for use by the session bean *SeqidSessionBean*. In a non-cluster environment, the EJB container synchronizes the concurrent access. But in cluster environments, the concurrent control is left to the PostgreSQL database. The default setting may give the chance for different session beans to fetch the same block number, which leads to create entity beans errors because of a duplicated key. Therefore, synchronization should be in each row in the table. Row level locking is used for the *SeqidBean* bean in this application.

```
<entity>
        <ejb-name>SeqidBean</ejb-name>
        <row-locking>true</row-locking>
        <table-name>seqid</table-name>
        ... ...
</entity>
```

Figure 52: Row Level Locking for SeqidBean Entity Bean

To prevent PostgreSQL becoming a bottleneck in an application, JDBC connections are another main factor that affect the performance of the application server and database. Essentially there should be enough connections between the application server JBoss and the database PostgreSQL for this application. The default database connection in PostgreSQL is 100. The value is too small for the stress test. In JBoss, the maximum connections are 200 for a single JBoss. Therefore, the connections number is increased to 1024. This connection number is enough for this test. Of course, increasing the connections in the PostgreSQL needs enough shared memory as discussed in section 4.2.1.

The study has attempted to cluster web server Tomcat and application server JBoss. For modern distributed system, database clustering will become necessary. The performance of a clustered database with various load-balancing policies is left for future work.

6.2.5 Test Plan Summary

The purpose of testing is to verify the performance of clustered JBoss changing due to applying various load-balancing policies at the EJB level. All experiment tests will run various open source software on different machines. The scenario is shown in figure 53. Variations exist in the result of each test so all tests are repeated three times.

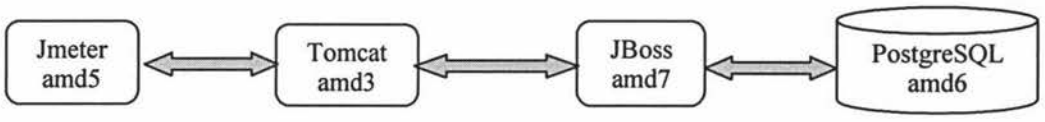


Figure 53: The Scenario of Testing Configuration

The entire test sequence of experiments is:

1. To determine ramp-up period and loop values
    a. For the test plan with thinking time;
    b. For the test plan without thinking time;
    c. Choose the appropriate version for the performance test;
2. To determine Tomcat level configuration for stress test
    a. Single Tomcat;
    b. Clustering two Tomcat with Apache;
    c. Directly interface to two Tomcat in Jmeter;
3. Scalability test
    a. Performance test for single JBoss;
    b. Performance test for two JBoss with default policy;
4. Comparison among various load-balancing policies effect on the performance of clustered two JBoss with equal machine load
    a. Performance test for two JBoss with default policy;
    b. Performance test for two JBoss with random robin policy;
    c. Performance test for two JBoss with first available policy;
    d. Performance test for two JBoss with first available proxy policy;
    e. Performance test for two JBoss with dynamic policy;
    f. Performance test for two JBoss with dynamic weight-based policy.
5. Comparison among various load-balancing policies effect on the performance of clustered two JBoss with unequal machine load
    a. Performance test for two JBoss with default policy;
    b. Performance test for two JBoss with random robin policy;

c. Performance test for two JBoss with first available policy;

d. Performance test for two JBoss with first available proxy policy;

e. Performance test for two JBoss with dynamic policy;

f. Performance test for two JBoss with dynamic weight-based policy.

Chapter 7: Test Results and Discussion

Summary: This chapter presents the details of the tests implementation on the Sisters cluster. The main tests are classified into two kinds – scalability tests and load-balancing policy comparison tests. The scalability tests refer to the comparison among a single JBoss and clustering two or more JBoss. The load-balancing policy tests will apply various load-balancing policies on the EJB level to identify if each of them affects the final performance of clustering JBoss.

All client nodes in the Beowulf cluster Sisters have slightly different CPU processing capability. In order to ensure all tests are under fair conditions, the same capability node is chosen to run the open source test software in each test. For example, the node amd3, amd4, amd7 and amd8 has the same setting and capability, Any of them can be freely selected to run JBoss or Tomcat that should not affect the final test result.

The software has been adjusted to adapt the stress test as discussed in previous chapters such as increasing connection numbers and allocating more JVM memory etc. On the other hand, before each test starts, the database is initialized to avoid overflow because every test would insert a large amount of data into database. Jmeter is also restarted to release memory between the tests. The test will follow the test sequence presented in section 6.2.5. Each test result is the average value from three repeated runs. The Jmeter will record the test results in an aggregation report and a graph result listener. The important parameters are recorded for later analysis including *number of total threads, average time delay, max time delay, throughput* and *hit rate*. Both *throughput* and *hit rate* reflect the performance of servers. The *throughput* of server increases due to increased requests rate. That is more requests generate more responses. The *average time delay* indicates the average waiting time of HTTP request for getting a response. Usually this is required to be less than 1 second for acceptable performance. And the *max time delay* field will record the longest waiting time for the HTTP request from all requests. For each test, the *max time delay* always get the highest variation since the EJB container may create different number of EJB instances for every test. The *max time delay* value should be less than 30 seconds because online customers may lose patience if waiting longer. Since the *hit rate* reflects the same performance as

throughput does, and *max time delay* only reflects one HTTP request, the *throughput* and *average time delay* are selected to analysis the performance of the entire system.

7.1 To Determine Ramp-up Period and Loops Value

The scenario of testing is shown in figure 53 in section 6.2.5. The stress test starts from Jmeter via setting three parameters, *number of thread groups, ramp-up period* and *loop values*. The testing cannot adjust all three parameters for each test since it leads to a large number tests. Since the number of thread groups directly indicates the number of clients in the test, it should be the most important parameter to affect the performance of the entire system. As a result, it is better to fix the ramp-up period and loop values for each test. Larger loop value also indicates longer experiment time. As many tests are run, each test should complete in 10 minutes.

Firstly the test plan included thinking time because it is more realistic to reflect clients' behaviors. Each thread group under the test plan has 21 threads (see figure 44) that may represent 21 independent clients. The total numbers of simulated clients participating in each test are:

```
Total clients = number of thread groups * 21 * loops
```

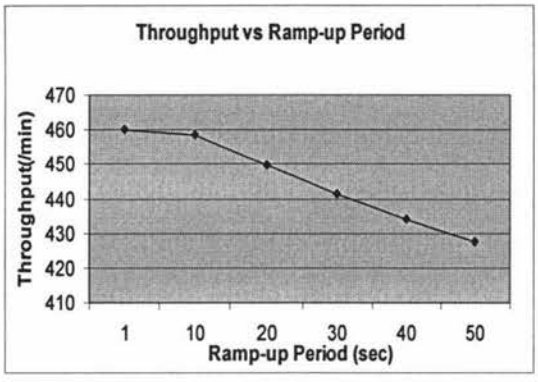
The table 2 shows the allocation of machine nodes in Sisters to participate in this parameter-setting test.

Software Participate in Test	Hardware	
	Run On	Single CPU Capability (MHz)
Tomcat	amd3	1733.745
JBoss	amd7	1733.745
PostgreSQL	amd6	1800.427
Jmeter	amd5	1800.427

Table 2: Machines Allocation for Jmeter Parameter-Setting Test

7.1.1 For the Test Plan with Thinking Time

The test plan with thinking time implemented in Jmeter gives each HTTP request a random processing time to simulate client's behaviors. For example, the client is allocated about 1 minute to complete the "price quote" form as a real client consumes time to fill a form.



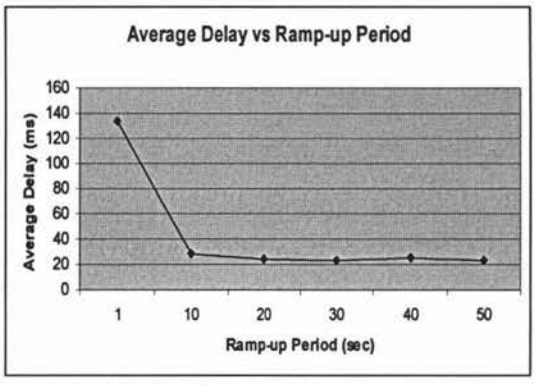


Figure 54: Ramp-up effect on Throughput **Figure 55: Ramp-up effect on Average Delay**

Since the requests hit rate on the server is too low with a small number of thread groups, the first experiment fixes the thread groups to 100. Each loop for the thinking time version spends about 5 minutes. Since many tests have to be done, it is necessary to shorten the testing time. A loop value of 2 is used to investigate how the ramp-up period (in seconds) affects the final performance.

The figure 54 shows the throughput of single JBoss will linearly drop after a ramp-up period value larger than 10 seconds. And figure 55 shows the ramp-up period effects the average time delay for all HTTP requests. Both measurements reflect the performance of the application server will be affected by the number of concurrent processing threads. The best result has the highest throughput and lowest average delay. It is obvious the ramp up period of 10 seconds is the best value.

Increasing the number of loops will result in longer processing time for each test. Figure 54 and 55 have presented the data for choosing ramp-up period and number of loops. The setting for each 100 thread groups unit will assign 10 seconds ramp-up period. That means the ramp-up period will be adjusted to 20 seconds if the number of thread groups is 200. Each test will keep the loop value to 2 for the thinking time version.

7.1.2 For the Test Plan Without Thinking Time

The test plan without thinking time is a very similar to with thinking time version except there is no random time delay for each HTTP request. The server process such requests quickly needing only a few second to complete all 21 HTTP requests in a thread group if all requests process concurrently. The throughput and hit rate are very high for this version. Therefore, the parameters setting should be different. Allocating 10 thread groups and 20 loops runs the same number of threads as the thinking time version. This test takes less than 3 minutes to complete.

The no thinking time version seems unstable since for each test the variation is higher than the thinking version. The figure 56 shows the throughput is unstable. When the ramp-up period is 2 seconds the server gives the highest throughput and lowest average time delay. The unstable result may come from the small number of loops. Since the processing speed is too fast for the no thinking time version, the server completes all requests before it reaches the stable period. Additional experiments were required to find a suitable loop value.

The numbers of thread groups are fixed to 10 and a 2 seconds ramp-up period is used. The number of loops is increased from 10 to 60 improving the result as shown on figure 58 and figure 59. Although 60 loops value presents the highest throughput and lowest delay, the experiment runs longer than 40 loops. And actually when the number of loops reaches 40, the throughput nearly gets the highest value and the average delay is also lower than 70ms. The experiment consumes about 8 minutes. It is the acceptable value for the test.

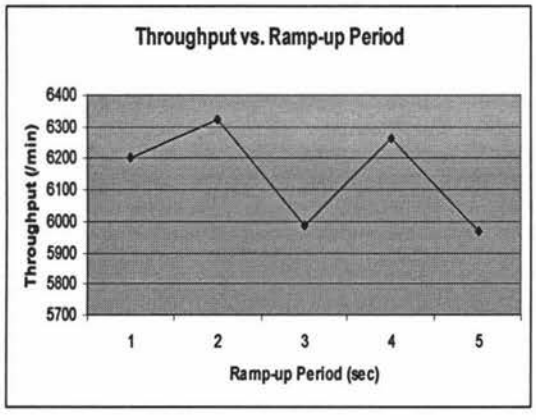


Figure 56: Ramp-up effect on Throughput

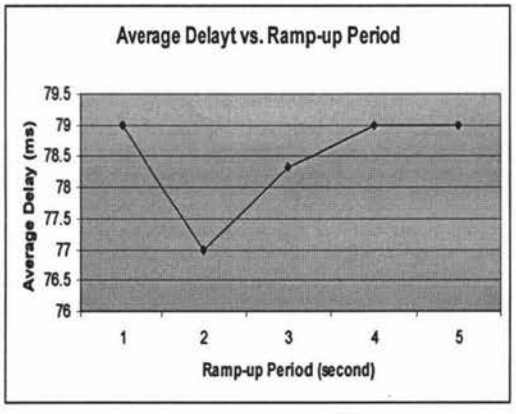


Figure 57: Ramp-up effect on Average Delay

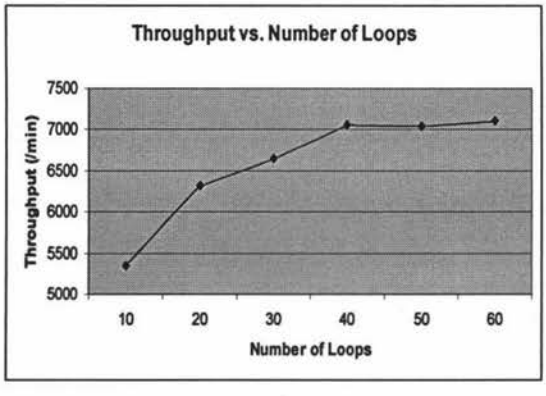


Figure 58: Loops Effect on Throughput

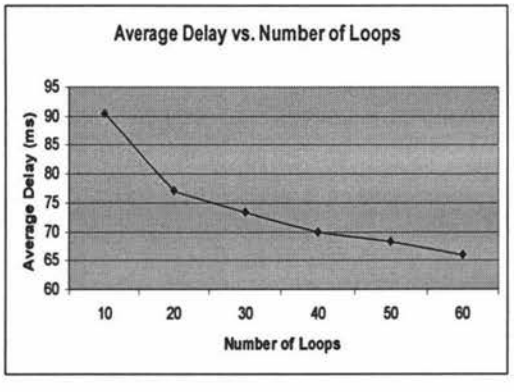


Figure 59: Loops Effect on Average Delay

Therefore, the parameter setting for no thinking time version is that for each 10 thread groups will assign 2 seconds ramp-up period and the number of loops are 40 for the later experiments

7.1.3 Choose the Appropriate Version for the Performance Test

The purpose of performance test is to find the maximum capability of JBoss. The thinking time version is more realistic but also gives a certain delay time to process each request. It should lead to process less requests in a time unit. The table 3 shows the performance result to process 4200 requests using the thinking time version and without thinking time version. We can see the hit rate for without thinking time version is nearly 14 times higher than with thinking time version per second. The same can be seen on throughput value. Therefore, to process the same amount of requests, the server result is lower for the thinking time HTTP requests. As a stress test to find the max performance of JBoss, it is not an ideal testing version.

	Total Threads	Average (ms)	Max (ms)	Throughput(/min)	Hit Rate (/sec)
Timer	4200	28	2265	458. 371	7. 6
No Timer	4200	76	3259	6338. 5063	105. 6

Table 3: The Comparison of Performance for Different Version

In addition, in order to get the larger amount of throughput, the only way is to increase the number of thread groups because the ramp-up period and loops value are fixed for every test. For the thinking time version, to increase the thread groups to 1000 gives a hit rate just around 50 (/sec). Meanwhile, the large number of thread groups will add a heavy burden on the Jmeter. Usually each Jmeter can handle 100-300 thread groups at most. Otherwise Jmeter will be a bottleneck in the test.

After comparison and consideration, the without thinking time version would be better for the stress test since it makes the server work harder. As a result, the no thinking time version is used for all the following performance tests.

7.2 To Determine Tomcat Level Configuration for Stress Test

The stress test is an n-tier system. The top level is Jmeter to simulate a large amount of requests on the second level – web server Tomcat. The purpose of the test is to test the performance of JBoss with various load-balancing policies at the EJB level. So except JBoss, all other software i.e. Jmeter, Tomcat and PostgreSQL should not be a bottleneck on the testing system.

The database PostgreSQL is configured with high capability to have a maximum of 1024 connections. This setting should avoid a connection bottleneck occurring on PostgreSQL since the maximum connection pool of JBoss AS is set to 200, i.e. the PostgreSQL can handle at least 5 JBoss AS concurrently. Of course, some critical tables or their fields in the database may become bottleneck such as Seqid table. The row-locking mechanism is used to control concurrent access to avoid a duplicated key error.

Although Jmeter can be configured to run in distributed mode to simulate higher stress, that requires one more machine to run Jmeter. This will reduce the number of machines available for the scalability test. In addition, all the distributed Jmeter will share the same test plan to communicate with the same web server Tomcat, which leads to higher stress on the back-end servers including the web server Tomcat. Therefore, the distributed Jmeter would cause the bottleneck problem on Tomcat level. Therefore, Jmeter is used in non-distributed mode to generate a high stress workload for the Tomcat level, which ensures all requests can be passed smoothly to the third level – application server JBoss (single or clustered).

7.2.1 Single Tomcat

A single web server Tomcat is placed in front of a single JBoss and clustered two JBoss servers to construct a stress test. If Tomcat is not a bottleneck in the testing system, the performance of clustered JBoss should be better than the single JBoss system with higher throughput and hit rate.

	Average (ms)	Throughput(/min)	Hit Rate (/sec)
singleTomcat->JBoss	546.667	10071.698	167.87
singleTomcat->TwoJBoss	560	9870.159	164.5

Table 4: Stress Test with Only One Tomcat

For the stress test, heavy loads of 100 thread groups are simulated in Jmeter. There are a total of 84000 threads involved in the test. The table 4 shows the test results from a single JBoss and clustered two JBoss with only one Tomcat. From the table, we see no better performance after clustering JBoss.

The parameters in table 4 shows, for single JBoss, the hit rate reflects the approximate throughput of 168 requests that the servers process per second. The average time delay for each request is about 547ms. For each test, the throughput is the actual throughput data for the duration of the entire test that is measured in Kilobytes per minute. The throughput and hit rate are the same thing but measured in different

way.

During the test, the Tomcat is busier with 84000 threads than JBoss (shown using the *top* command in Linux). Since sometimes the CPU usage on running Tomcat machine reaches 100% that means the bottleneck has been reached on the Tomcat side. From the table 4, we see the performance of JBoss does not improve with cluster of two JBoss but also have dropped a little. The jobs are shared well in the clustered environment but the overall performance does not increase.

With the same loads, why the performance of clustered JBoss is a little bit worse may be for various reasons, such as JBoss has to keep the state for stateful session bean that requires extra communication among clustered member. It also may be caused by the backend database control when two EJB containers try to access the table concurrently. If it is the case, all the later experiments will have the same concern since the test architecture is constructed as a central distributed mode using a shared database.

7.2.2 Clustering Two Tomcat with Apache

The first stress test encounters the bottleneck problem using a single Tomcat. To increase the number of requests from the Tomcat level, clustered Tomcat may be a solution. As stated in section 3.2.3.1, Tomcat clustering needs Apache HTTP Server and a load balancing mechanism. The testing architecture actually adds one more application server Apache with mod_jk in front of the two Tomcat. All requests sent from the Jmeter pass to Apache server instead of passing directly to Tomcat.

The test uses 100 thread groups in Jmeter. If the load balancer mod_jk works perfectly, then each Tomcat will be allocated 50 thread groups. Such setting may remove the bottleneck on Tomcat level, but it also possibly introduces a new bottleneck level – Apache.

However, during the test, the Apache server dispatches HTTP requests with a slow speed and each test runs longer. Once the HTTP requests are overloaded, some HTTP requests get error responses code 503. Testing 100 thread groups; every time there were about 30% error HTTP responses. The sample error response messages see the following.

```
HTTP response headers:
HTTP/1.1 503 Service Temporarily Unavailable
The server is temporarily unable to service your request due to maintenance downtime or capacity problems. Please try again later.
```

In turn testing small number of threads with Apache i.e. only 10 thread groups, usually there are no error HTTP responses. But the performance of the entire system is poor. The throughput is very low compared to without Apache configuration (see table 5).

Communication Way	Average (ms)	Throughput(/min)
jmeter -> JBoss	95	5327.8328
jmeter -> tomcat1 ->JBoss	80.67	6376.929
jmeter -> apache -> tomcat1 ->JBoss	438.3	1323.172
jmeter -> apache -> tomcat1 + tomcat2 -> JBoss	440.67	1323.896

Table 5: Performance Comparison among Various Web Level Configurations

From the table 5, we can see the system presents the best performance after separating the web server Tomcat from JBoss but without Apache. The HTTP server Apache give a significant drop in the performance of the entire system even with clustered web servers Tomcat. In fact, the CPU usage of Apache running machine did not reach 100%, at most time it was around 50%. The explanation for such output is that the Apache controls the dispatching of HTTP requests and it only accepts a certain amount of HTTP requests in a time unit. If the amount exceeds the maximum value, the server refuses to service and returns an error message instead. Therefore, such web level configuration is not suitable to test the performance of the application servers.

7.2.3 Directly Load Two Tomcat in Jmeter

The third solution is to create two thread groups under the same test plan in Jmeter. The second thread group just simply copies the first one but with different HTTP request defaults. The HTTP request defaults define the IP address and listening port of the Tomcat server. That means HTTP requests are configured in different thread groups to dispatch to different web servers. Therefore, two Tomcat servers are run in two machine nodes, i.e. Tomcat1 on amd3 with port 8081 and Tomcat2 on amd4 with port 8082 (Tomcat1 and Tomcat2 can have the same port number if they run on different machines). These two Tomcat are not configured with state replication as in Tomcat clustering, since all HTTP requests can be passed smoothly to simulate heavy loads on the next level – JBoss. In Jmeter, the first thread group is configured to send all HTTP requests to Tomcat1 and all HTTP requests in the second thread group will be passed to Tomcat2. The architecture is illustrated in figure 60.

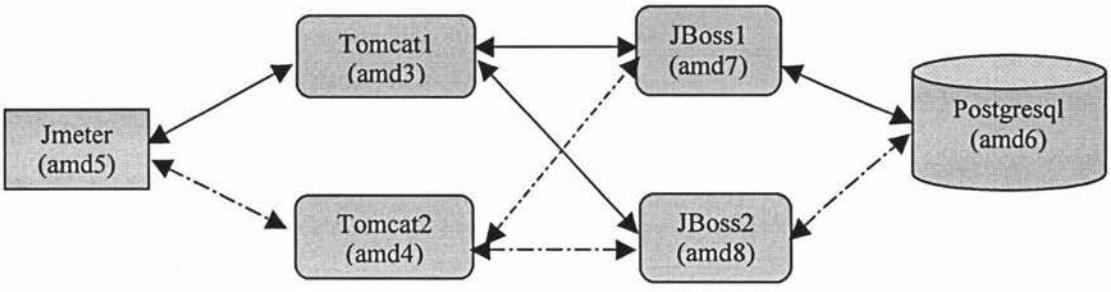


Figure 60: The Architecture of Stress Testing System

For such a configuration, a total of 100 thread groups are used as before, but this time, the Tomcat1 and Tomcat2 share the total workloads and get 50 thread groups each. Such configuration removes the overload on the Tomcat level and it does not add extra burden to Jmeter. The test proved a better performance gain after JBoss clustering. The table 6 shows the results for 100 thread groups on a single JBoss and clustered two JBoss system.

	Average (ms)	Throughput(/min)	Hit Rate (/sec)
For single JBoss	522	10627.69	177.1
For Two JBoss	413	12582.03	209.47

Table 6: Stress Test with Starting Two Tomcat from Jmeter

The table 6 presents the higher throughput and shorter waiting time for each HTTP request can be obtained after JBoss clustering. However, the performance has not significantly improved, i.e. the performance of clustered two JBoss does not obtain twice that of a single JBoss. It maybe the bottleneck occurs on Jmeter or PostgreSQL. But at least such web level configuration has removed the bottleneck on Tomcat level and improves. All the following experiments will use the architecture shown on figure 60.

7.3 Scalability test

After completing the first stage configuration including the selection of test plans, determining testing parameters values and Tomcat level configuration, the critical performance tests including scalability tests and load-balancing policy tests where conducted.

The scalability test refers to higher performance obtained due to adding one or more redundant server nodes to the original server system. How the performance of clustered two JBoss servers is better than a single JBoss system is examined. The architecture of

Software Participate in Test		Hardware	
		Run On	Single CPU Capability (MHz)
Jmeter		amd5	1800.427
Web Server	Tomcat1	amd3	1733.745
	Tomcat2	amd4	
Application Server	JBoss1	amd7	1733.745
	JBoss2	amd8	
PostgreSQL		amd6	1800.427

Table 7: Machine Allocation for Scalability Test

the testing system uses centralized workload configuration i.e. all JBoss shared the single database PostgreSQL. All software runs on independent machines without extra loads. Table 7 gives the basic allocation of machine nodes for the software involved in the scalability test.

From table 7, we can see the highest capability machine nodes in Sisters have been allocated to the Jmeter and PostgreSQL, which are the possible bottleneck sources for the current testing system.

7.3.1 Performance Test for a Single JBoss

For the single performance test, there is only one JBoss server involved in the stress test. That means only one EJB container involved in the test. Then the EJB container will be responsible to manage the concurrent access to keep the data consistence. In addition, the EJB container also maintains the JDBC connection pool to the PostgreSQL database. Checking connections via the command *ipcs –m* shows there are initially 50 connections established when the JBoss is ready. During the test, the number of connections increases while the number of HTTP requests increases. For 100 thread groups, the dynamic connections can reach up to 101 during the test. This number is not static for every test. The EJB container will create new connections depending on the number of concurrent threads. Without resetting the size of the connection pool, the previous tests often encounter "*NoConnectionManager available*" exception. But the connections bottleneck has been removed after increasing the size of connection pool.

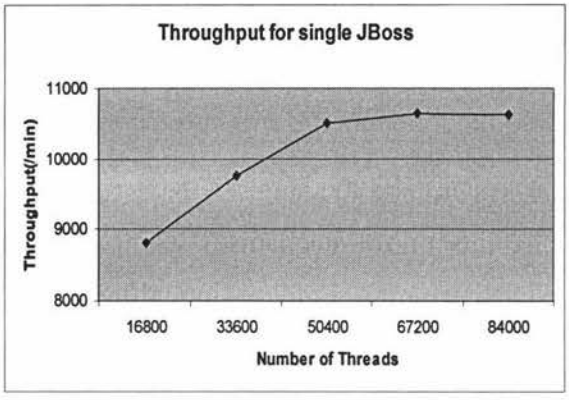


Figure 61: Throughput from Single JBoss

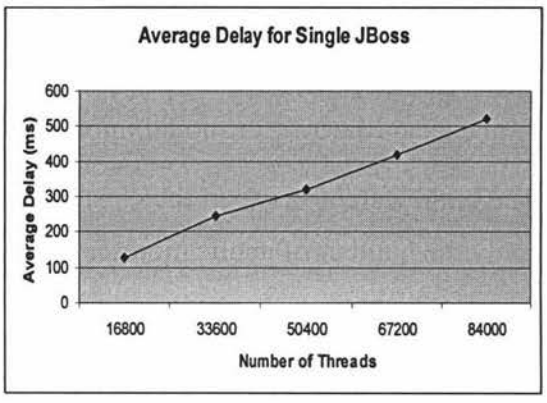


Figure 62: Average Delay from Single JBoss

The previous experiments have found the optimal loop value and ramp-up period for each 10 thread groups. So the performance test of single JBoss will directly increase the number of thread groups from 10 to 50 for both Tomcat1 and Tocmat2 and their ramp-up period from 2 to 10 in the Jmeter. The Jmeter listeners will record the total performance e.g. throughput and average time delay for the entire test system. The test results shown on figure 61 - 62.

The figure 61 shows the final throughput under the varying load i.e. processing 16800 threads to 84000 threads in a short period. The throughput is linearly increasing with the number of threads less than 50400. The throughput of a single JBoss system has increased slightly when the number of threads is over 50400. But the maximum time delay is still linearly increasing (see figure 62). It indicates the processing speed and ability of single JBoss system has nearly been saturated i.e. the bottleneck of single JBoss system is nearly reached. Each HTTP request will wait longer if the system handles the total requests over 50400.

During the test, the CPU usage for each running machine was checked. With the heavy HTTP requests coming from the two Tomcat, the single JBoss is busy most of time. The percentage of CPU idle on the JBoss running machine often reaches 0. The machine running Jmeter is busy sometimes. Since the throughput reflects the performance of entire system, this bottleneck may occur on the JBoss or Jmeter. It is also possible from the backend database server.

7.3.2 Performance Test for Clustered Two JBoss

The second stage of the scalability test is to add one more JBoss into the testing system. Two clustered JBoss servers use the default load-balancing policy – round robin. The architecture of the testing system is shown in figure 60. In the single JBoss testing

system, the bottleneck of single JBoss system is reached when the system handles 50400 threads in a short period. If the bottleneck comes from the JBoss, then the performance of the system should improve after adding one more JBoss into the system.

The big different between the clustered JBoss system and the single JBoss system is not only the number of application servers but also the control mechanism. Compared to the single JBoss system, the JBoss clustering is more complex including state replication and data persistent maintenance. The concurrent control mechanism of sharing data will be left to the backend database server when the application servers are clustered. In general, a more complicated system usually requires more communications.

The initial JDBC connection pool size for the clustered two JBoss is double that of a single JBoss system. It is 100 in this test. Except the communication between the clustered members, the performance of clustered two JBoss may be nearly double of a single JBoss system too if all other applications are not the bottleneck in the system. The figures 63 – 64 presents the experiment results of clustered two JBoss system.

The throughput from clustered two JBoss system on figure 63 presents better curve for number of threads from 50400 to 84000 than from the single JBoss system (compare with figure 61). It indicates the clustered two JBoss system has better capability to handle more connection threads.

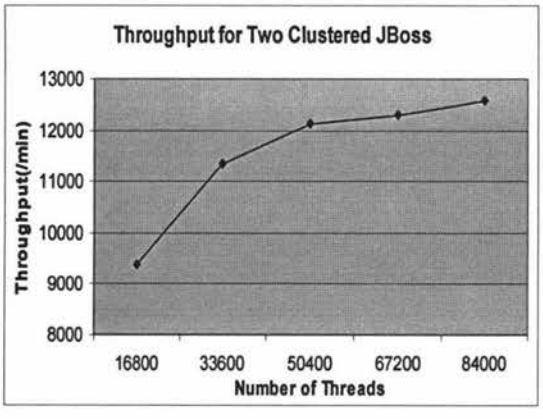


Figure 63: Throughput from Two JBoss

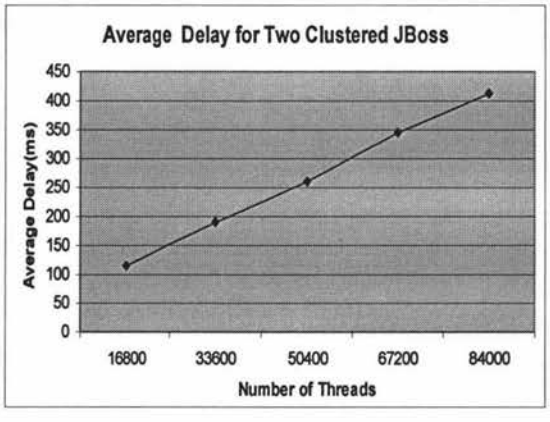


Figure 64: Average Delay from two JBoss

7.3.3 Comparison and Discussion

The figure 65 shows the throughput comparison of both testing systems and figure 66 presents the average time delay for the HTTP requests from the scalability testing systems.

In figure 65, the lower curve represents the output from the single JBoss system. It is obviously the throughput of the clustered two JBoss system is better i.e. it has higher capability to handle the same amount of requests than the single JBoss system. The better performance is also reflected in the figure 66, which indicates the average time delay for both testing system is linearly increasing with the increasing number of threads. But compared to the single JBoss system, the clustered two JBoss has lower average time delay (see figure 66 lower line) for the HTTP requests after increasing the number amount of requests. In addition, from the figure 65, the two lines have diverging gradient with increasing the number of threads, which indicates the HTTP requests should wait longer to get responses in a single JBoss system while the system processes more and more requests.

The table 8 lists the quantitative measurement for both testing systems. The parameter oneTP represents the throughput of the single JBoss system and twoTP is the throughput of the clustered two JBoss system. The table indicates that to handle 16800 threads i.e. 16800 HTTP requests; the two JBoss system has higher 5.85% capability than the single JBoss system. This capability increases up to around 13% -15% when the system handles more than 33600 requests.

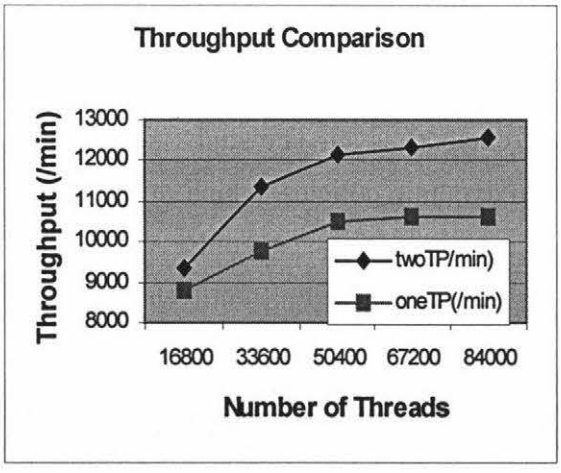


Figure 65: Throughput Comparison

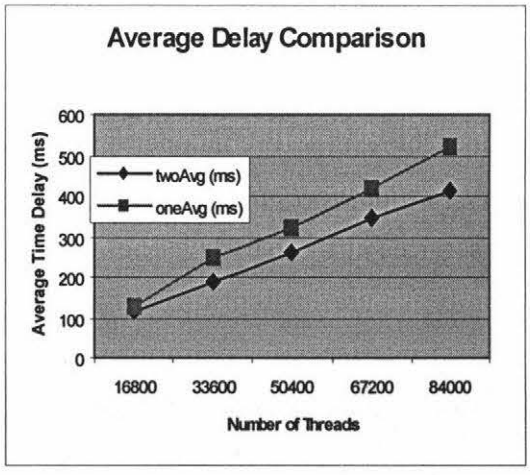


Figure 66: Average Delay Comparison

Total Threads	oneTP(/min)	twoTP(/min)	Percentage (%)
16800	8815. 66	9362. 934	5.85
33600	9749. 725	11337. 313	14.00
50400	10502. 982	12139. 156	13.48
67200	10637. 446	12300. 406	13.52
84000	10627. 693	12582. 029	15.53

Table 8: The Quantitative Analysis for Scalability Testing Systems

Since more communications and control mechanisms are needed for a clustered system, the system usually presents worse or a little bit better performance than the single application server system if the clustered system handles fewer amounts of requests in a short time period. But the situation will improve after increasing the number of requests because the longer running time makes the results more stable. That is why the throughput of the two JBoss system is only 5.85% higher than single JBoss system for 16800 threads and improves for more than 33600 threads.

However, from table 8, we also notice the performance of two JBoss is not nearly double that of a single JBoss system but only about 15% improved. That means adding one more JBoss into the testing system has removed the bottleneck on single JBoss, but it cannot remove the bottleneck on other applications, e.g. Jmeter or PostgreSQL. The most probable bottleneck is on the database PostgreSQL since the testing application is sensitive to sharing a single table 'seqid' when creating an entity bean. To cluster the database may be a solution or creating an entity bean without primary key value as discussed in further work in Chapter 8.

7.4 Testing on Equal Machine Load

The load-balancing policy in a clustered J2EE application can be used at the web container level, at EJB container level and at the database level. The study is to extend JBoss *LoadBalancePolicy* interface to design two dynamic policies, which utilizes EJB timer services at the EJB level i.e. for session beans and entity beans only. Meanwhile session beans are in front of entity beans in this application and all entity beans have only local interfaces for performance reason. As a result, the performance testing will apply various load-balancing policies on all session beans.

The testing architecture and procedures are the same as for the scalability test with clustered two JBoss. The experiments run all applications on machines without loads.

7.4.1 Performance Test for Two JBoss with Built-in Policies

The JBoss built-in load-balancing policies as discussed in section 3.2.4.2 are *RoundRobin, RandomRobin, FirstAvailable* and *FirstAvailableIdenticalAllProxies*. The load-balancing policy is applied to the home and component interfaces of all session beans in the MGProject application. For stateless session beans i.e. *custSessionBean, orderSessionBean, discountSessionBean, priceQuoteSessionBean* and *SeqSessionBean,* both home and component interfaces are assigned the same policy for my experiments. For example, when the experiment tests the *RoundRobin* policy effect on the performance of system, then clients are told to use *RoundRobin* policy to look up home interface of the session bean and use the same policy to invoke the methods defined on its component interface. For the stateful session bean – *orderCartBean,* the home interface of the stateful bean is assigned the same policy as the stateless session beans depending on the experiment. But the component interfaceis always assigned *FirstAvailable* policy since the server uses the sticky sessions to maintain the client's state. That means when the client looks up the instance of bean's home interface on *amd7* the first time, then the client is sticky on *amd7* invoking methods on the component interface too.

Each experiment is repeated three times. Figures 67 and 68 displays individual result from four built-in policies.

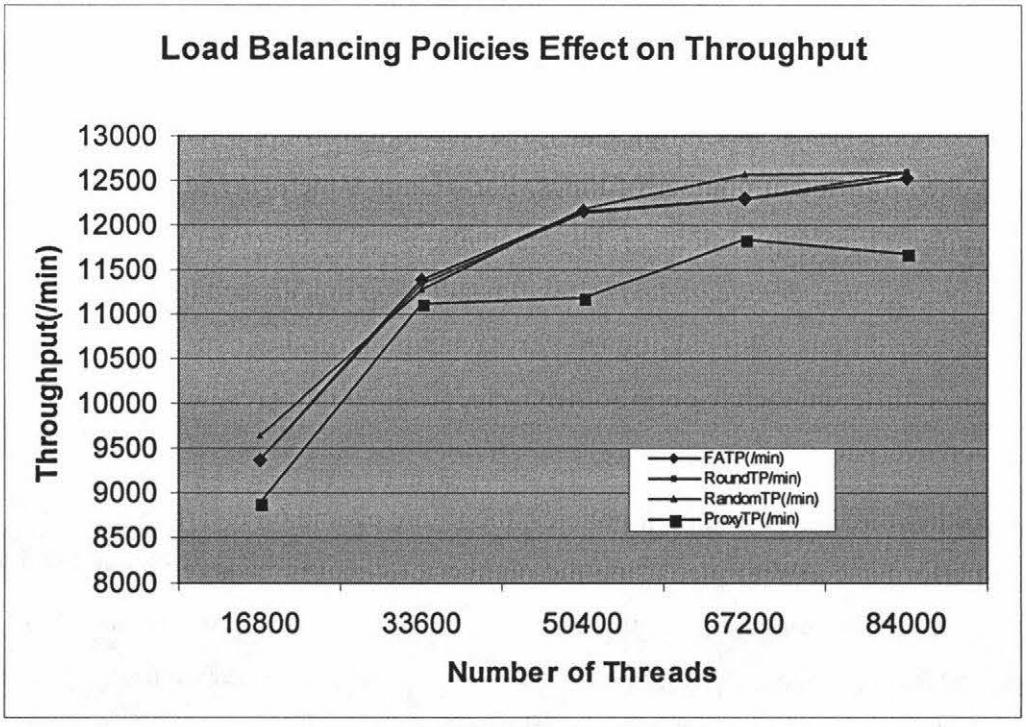


Figure 67: Policies Effect on Throughput

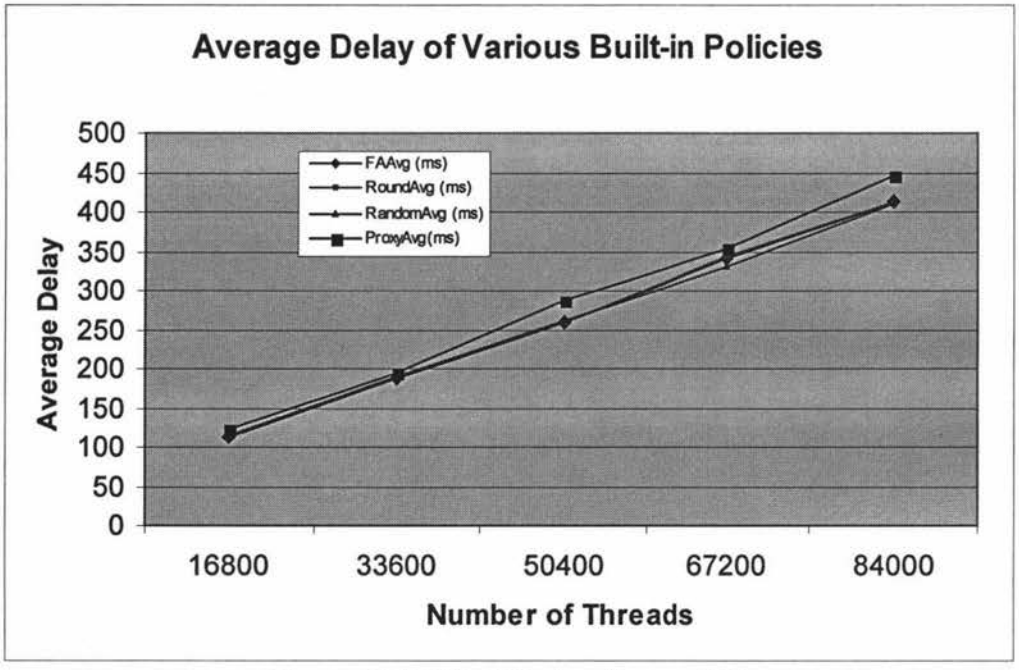


Figure 68: Policies Effect on Average Delay

On figure 67, the lowest curve is the output of the system with *FirstAvailableIdenticalAllProxies.* And the other three policies *FirstAvailable, RoundRobin* and *RandomRobin* have similar throughput that are hard to distinguish.

The figure 67 indicates the *FirstAvailableIdenticalAllProxies* policy presents the worst performance in clustered balanced machine-load environment compared to the other three policies. The difference of *FirstAvailableIdenticalAllProxies* policy is that all clients targets the same EJB in the given cluster belonging to the same proxy family that shares the same server. The policy aims to remove the side effect on invoking the home and component interfaces if the given EJB are in the different families. But every EJB is invoked a different number of times. For example, the *orderSessionBean* is the busiest bean in this application. If all calculation tasks of *orderSessionBean* are completed only at one server e.g. *amd7*, it will result in an unbalanced load and lead to a decrease in performance. In addition, every repetition of the experiment the EJB container has different behaviour. So from the testing data result, there are high variations on every test.

For the other three more common policies, the figure 67 and 68 clearly present nearly the same performance. With increasing the number of client threads, either throughput or average time delay for HTTP requests are nearly the same under the various built-in policies. The *RandomRobin* policy has a little bit better performance when the system is overloaded but may not be statistically significant.

However, observing the CPU usage and load average values, the different policies have different behaviors. The *RoundRobin* policy presents the best balancing during the test based on the percentage of CPU usage and the values of load average. And the *FirstAvailable* policy and the *RandomRobin* policy sometimes make one machine have a very high CPU usage and long job queues. But eventually it tends to balance with reasonably long loops.

The experiments of the load-balancing policy use the same architecture as the scalability experiments with clustered two JBoss, which has removed the bottleneck on JBoss level. But the system does not remove the bottleneck on other applications. Applying the *FirstAvailable* and *RandomRobin* policies to the system, the performance of the system nearly does not change. Therefore, the test results also indicate that the load-balancing policy *RoundRobin* is not a bottleneck of the system in the scalability test.

7.4.2 Performance Test for Two JBoss with Dynamic Policies

The two dynamic load-balancing policies have been designed for the performance test. Both dynamic policy and dynamic weight-based policy utilize the EJB timer service to dynamically load the information of the machine load average values every 30 seconds. For the dynamic policy, all HTTP requests are dispatched to the current idlest JBoss server within every 30 seconds. But for dynamic weigh-based policy, all HTTP requests are dispatched to all JBoss servers based on a calculated ratio within every 30 seconds.

The performance test results are shown on figure 69 and 70.

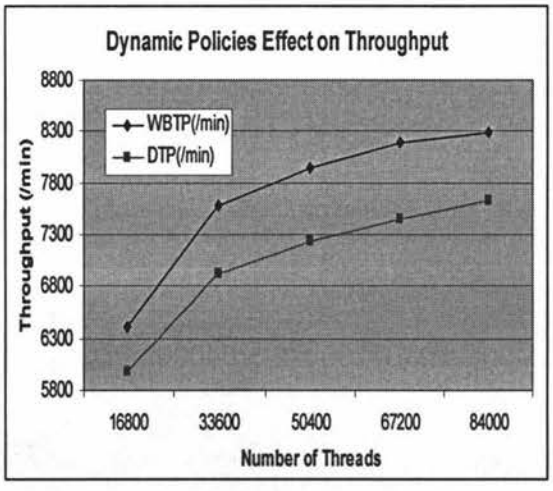


Figure 69: Policies Effect on the Throughput

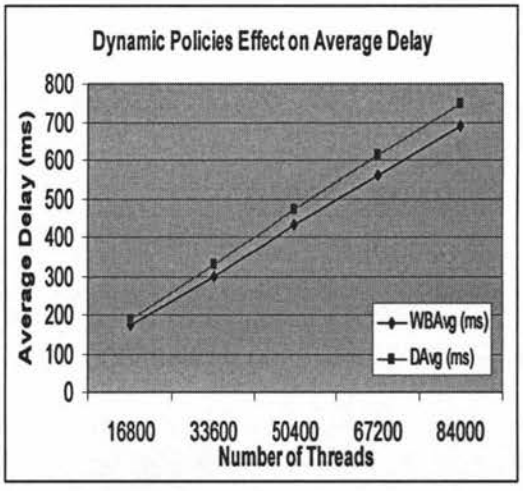


Figure 70: Policies Effect on Average Delay

Compared to the dynamic load-balancing policy, it is obvious the dynamic weight-based policy makes the system present better performance with higher throughput and lower average time delay for HTTP requests (see figure 69 and 70). The dynamic weight-based policy requires all servers to respond to the HTTP request at least once in every round depending on the calculated ratio value. However, the dynamic policy always requires only one JBoss server working during every 30 seconds. Faced with heavy client requests, the working JBoss may overwork and become the busiest node in the cluster during the 30 seconds, which leads to bad performance. Therefore, the dynamic policy cannot optimize the capability of every server in the clustered environment.

7.4.3 Comparison and Discussion

The experiments examined six load-balancing policies effect on the performance of an entire system on a balanced machine-load environment. The built-in policies except the *FirstAvailableIdenticalAllProxies* policy have similar good performance. And the dynamic weight based policy presents better performance improvement compared to the dynamic policy.

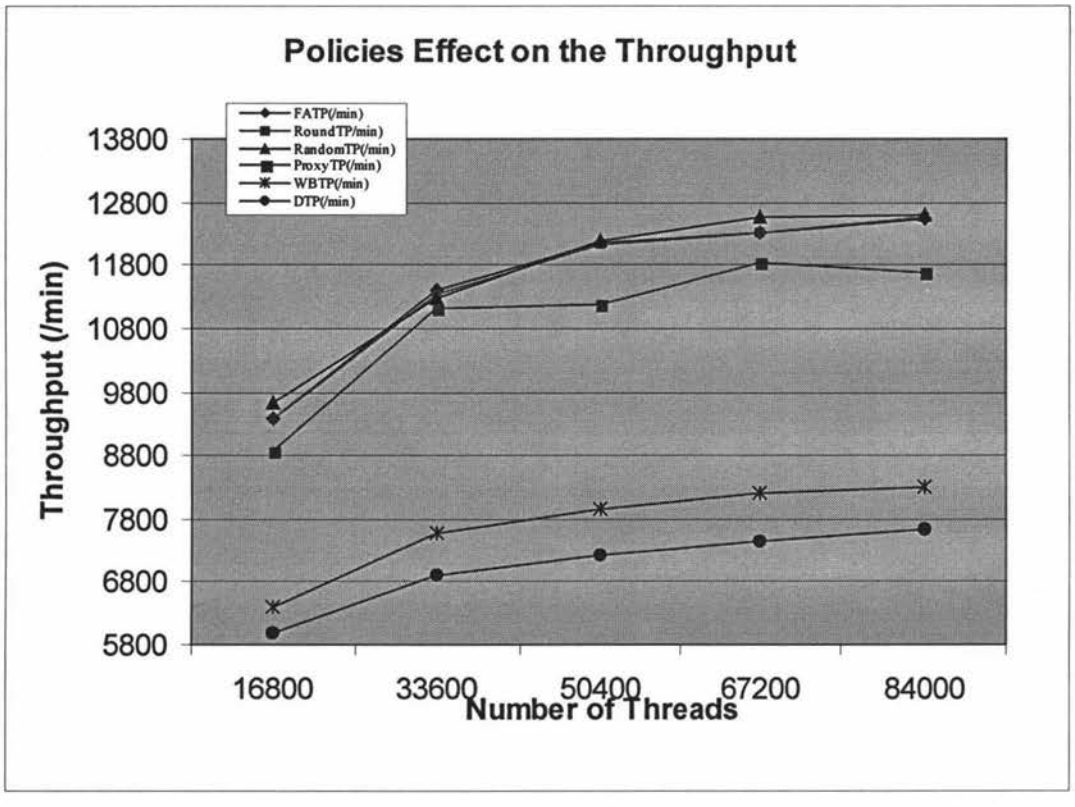


Figure 71: The Policies Effect on the Throughput without Extra Load

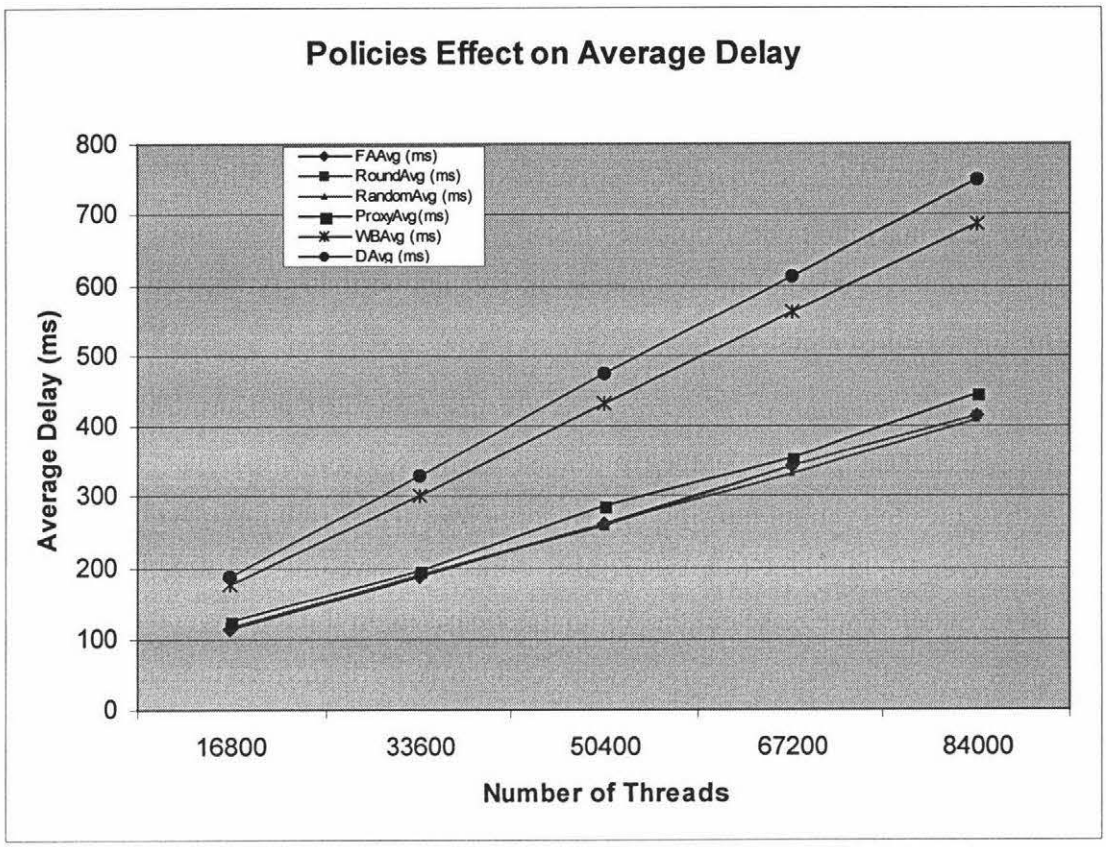


Figure 72: The Policies Effect on the Average Delay without Extra Load

The figure 71-72 clearly shows the four built-in load-balancing policies have significantly better performance compared to the two dynamic load-balancing policies either from throughput or from the average time delay for the HTTP requests. Such an outcome seems unexpected from the theory since the dynamic policies design are based on the load information about the JBoss server machines. The dynamic policies actually know which is the idlest JBoss running machine is and dispatch the requests to it. After analysis, there are two main reasons to cause the significant drop in the system performance for dynamic policies.

- *Extra step to lookup timer session bean plus additional calculations.* The dynamic policies need to invoke the session bean "*tmSessionBean*" and utilize the EJB timer service to read in system load average values in an interval period i.e. 30 seconds in my case. Every time after time out, some calculations have to be implemented to find the best machine node that will be selected to serve client's requests (for the details see section 5.2 & 5.3). Therefore, for JBoss with dynamic policies, the clients have to invoke the "*tmSessionBean*" via the EJB container at the first step and then get optimal dispatching information. Although current computers' CPU processing speed are very fast, if many servlets need to

invoke all session beans either on the home interface or on a component interface, they have to invoke the EJB container to get an instance of "*tmSessionBean*" first. This extra step plus the additional calculations give the extra burden to the heavy load JBoss. Unlike the dynamic policies, the three built-in policies just directly choose a machine node from the clustered members list based on the selection algorithm. There is not any extra step or calculation that needs to be performed for these built-in policies.

- *Load average values cannot exactly reflect the CPU usage*. During the test, the CPU usage and load average values were checked via the Linux commands. Sometimes the variation of job queues in the two JBoss running machines is very high over 10, but the CPU usages of two machines are only a little bit different. The dynamic policies select the optimal node based on the load average values of the system. If the load average values cannot exactly reflect the current CPU usage, this means the dynamic policies may select the unsuitable node.

The above comparison is based on the application server JBoss running on the machines without extra loads. Both JBoss server machines have equal capability. Such an environment is more suitable for built-in policies. If the JBoss server runs on unequal-load machines, the dynamic policies may optimise its function. Therefore, the following experiments will run the JBoss on unequal-load machines and do the same kind of comparison of the system performance with the six various load-balancing policies.

7.5 Testing on Unequal-Load Machines

In order to produce an "unequal" machine-load between *amd7* and *amd8* that the two JBoss servers run on, another small application - *random calculation* is started in *amd8* and run infinitely, which makes the CPU usage of *amd8* only 50% at the beginning of each experiment.

The experiments on unequal machine-load environment actually are the same as previous experiments on balanced machine-load either on the architecture or on the test procedures. The only difference is that *amd8* only 50% CPU is available when an experiment starts. This usage will change during the testing because the allocation of CPU depends on the job queue.

7.5.1 Performance Test for Built-in Policies

The four built-in policies on unequal machine-load environment present different behaviors compared to the balanced machine-load environment. That is shown on figure 73 & 74.

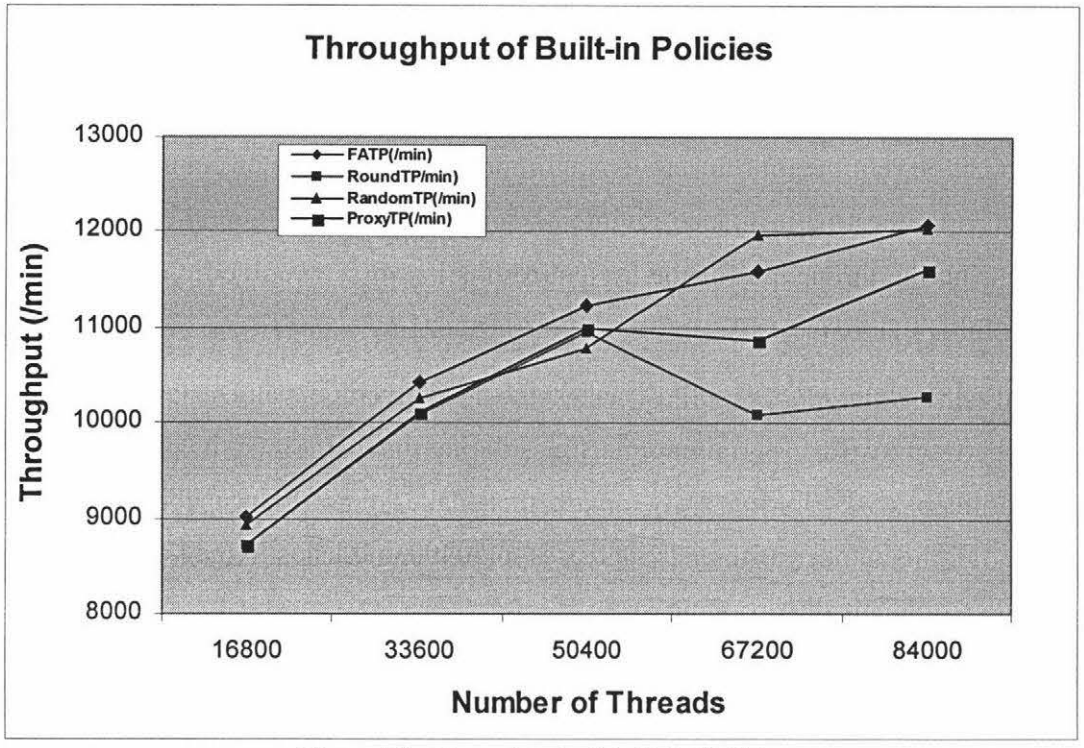


Figure 73: Throughput of Built-In Policies

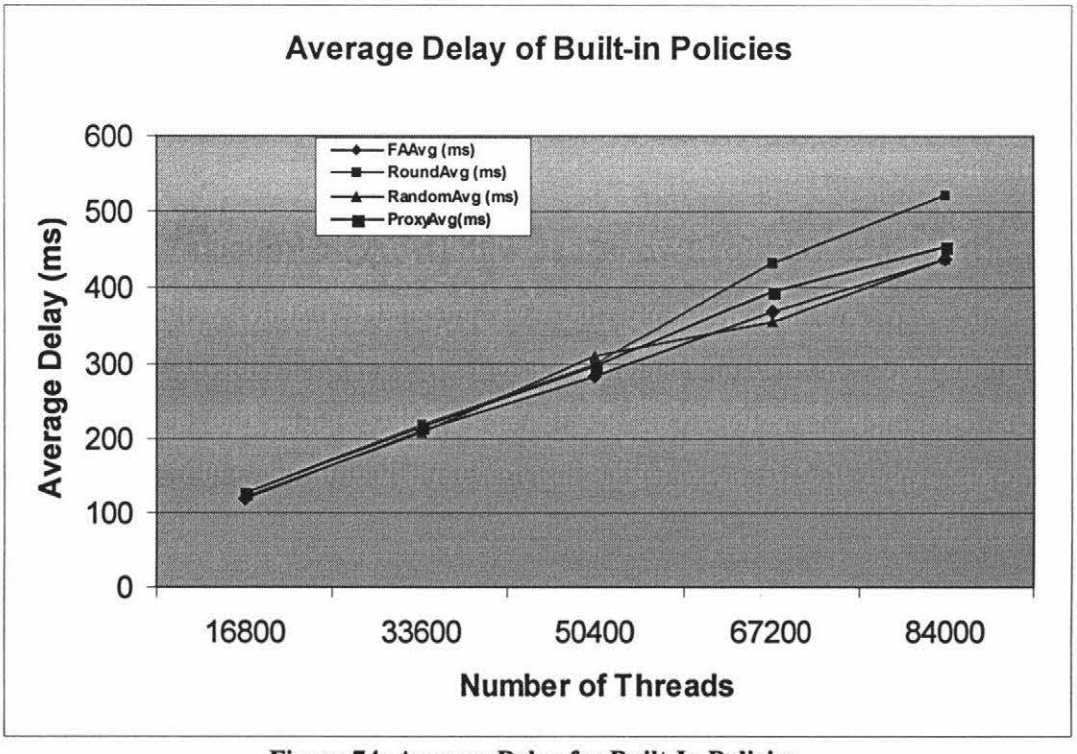


Figure 74: Average Delay for Built-In Policies

From figure 73, four built-in policies have a similar performance before the number of threads is up to 50400, which is nearly a bottleneck of the system as in the balanced machine loads environment (sees the figure 67 in section 7.4.1). After then, the different policies present different behaviors. The *RoundRobin* policy has the worst performance in an unequal environment when the system handles the requests over 50400. Next worse is the *FirstAvailableIdenticalAllProxies* policy. The *RandomRobin* policy presents better performance than the *FirstAvailable* policy around the number of threads reach to 67200. From the four unstable pattern for built-in policies on figure 73, the first available has the best performance giving a smooth curve. The figure 74 proves again that increasing the throughput of the system results in worse average time delay to get responses for HTTP requests.

Actually the *RoundRobin* policy makes the load average values have very high variation between two JBoss running machines during the test. Since the policy requires equal sharing of tasks for *amd7 & amd8*, this makes the CPU capability of *amd8* overload after the number of threads increases to 50400 and leads to the performance of entire system dramatically dropping.

For the other three policies, basically all of them use a random algorithm to select a server. So if the first time more threads select *amd7*, the performance of entire system is better. The performance of the system is not guaranteed due to the random selection. This can be viewed from the test data, repeated experiments have a big difference in performance, e.g. for one test the hit-rate is 200 but another output is only 150. That is why the graph is not smooth for these policies on the unequal machine-load environment.

7.5.2 Performance Test for Two JBoss with Dynamic Policies

Here, the experiments will apply the dynamic and dynamic weight-based load-balancing policies on the unbalancing machine loads environment. The dynamic weight-based policy is better than the dynamic policy under the balanced testing environment. Table 9 & 10 presents the difference of final throughput from both kinds of experiments.

For Dynamic Policy

Total Threads	DTP_noLoad(/min)	DTP_Load(/min)	Difference (%)
16800	5986.6	5881.66	-1.75
33600	6918.83	6984.65	0.95
50400	7233.75	7220.52	-0.18
67200	7451.12	7382.63	-0.92
84000	7626.83	7552.66	-0.97

Table 9: The Throughput Comparison for Dynamic Policy

Table 9, *DTP_noLoad* represents the throughput of the dynamic policy on the balanced machine-load environment. And *DTP_Load* is the throughput from the experiments on unequal machine-load environment. The difference refers to the percentage of throughput decrease on unequal machine-load compared to the balanced machine-load environment.

The table 9 indicates the performance of the entire system has only a little drop for clustered JBoss with dynamic load-balancing policy on unequal machine-load than on balanced machine-load environment. While the number of threads increases, the performance of the system on unequal load environment with dynamic load-balancing policy is better and close to the performance on the balanced machine-load environment.

For Dynamic Weight-Based Policy

Total Threads	WBTP_noLoad(/min)	WBTP_Load(/min)	Difference (%)
16800	6402.14	6001.9763	-6.25
33600	7585.623	7270.28	-4.16
50400	7947.4865	7823.78	-1.56
67200	8196.184	8048.42	-1.80
84000	8291.558	8239.216	-0.63

Table 10: The Throughput Comparison for Dynamic Weight-Based Policy

Table 10 shows the throughput decreases on the unequal machine-load environment for the system with dynamic weight-based load-balancing policy. But the situation gets better and better when the system needs to handle more and more HTTP requests. The performance is nearly the same as on the balanced machine-load environment when the number of threads is over 84000.

Both table 9 and table 10 shows the performance of the entire system with dynamic policies present better performance on unequal machine-load environment while the system face the overload requests.

However, the experiment proves again the load average values of machine cannot

exactly reflect the CPU usage. During the dynamic policies testing, the load average values are closer at most time, but it does not mean the CPU usage values are close too.

7.5.3 Comparison and Discussion

Figure 75, the best throughput for the unequal machine-load environment is from the built-in policies and the lower two lines refer to the throughput of dynamic policies. The figure shows the performance of the entire system is better with any built-in load-balancing policies than with dynamic policies. The figure 76 proves the same result but from the shorter average delay. The issues are the same as discussed on section 7.4.3.

However, the performance of the system with dynamic policies on unequal machine-load environment is good as on balanced load environment. The dynamic policies actually know the current machine load and balance to share tasks among the clustered members. Even the unbalancing environment, the performance of the system is better with increasing number of requests. The four built-in policies degrade in an unequal load environment. The four built-in policies become the bottleneck of the system while the number of request over 50400 for the unequal load environment. The shortcoming of the dynamic policies is that they involve too many communications and calculations, which give an extra burden to a heavily loaded application server that lead to a drop in the performance.

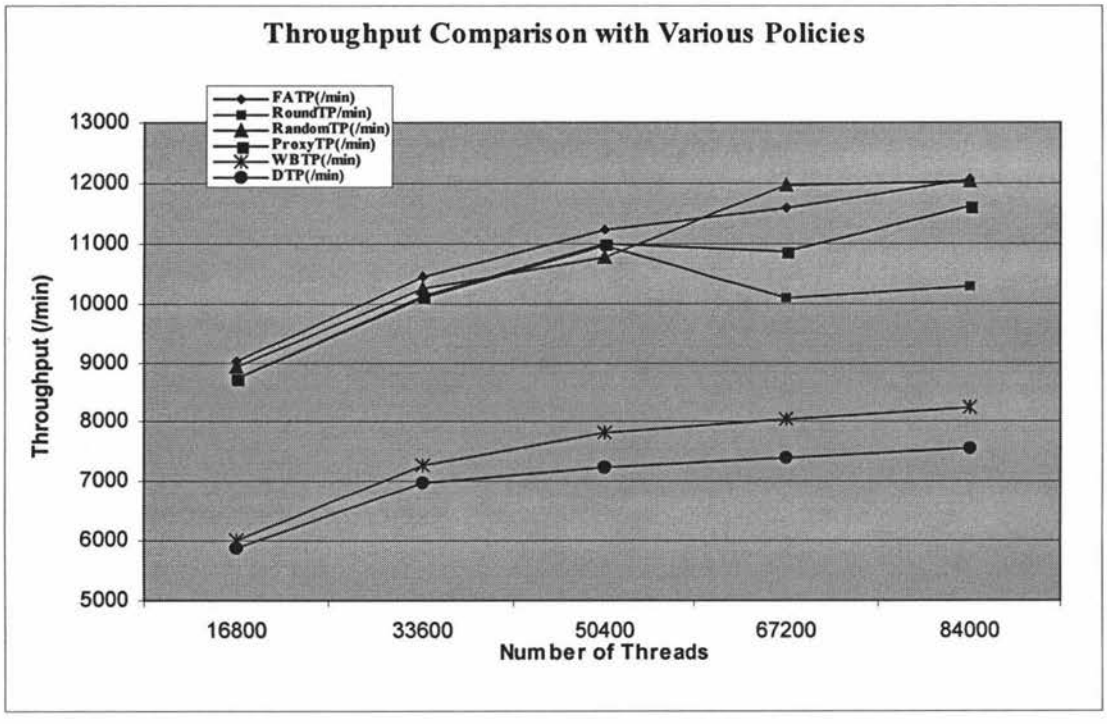


Figure 75: Throughput of Various Policies

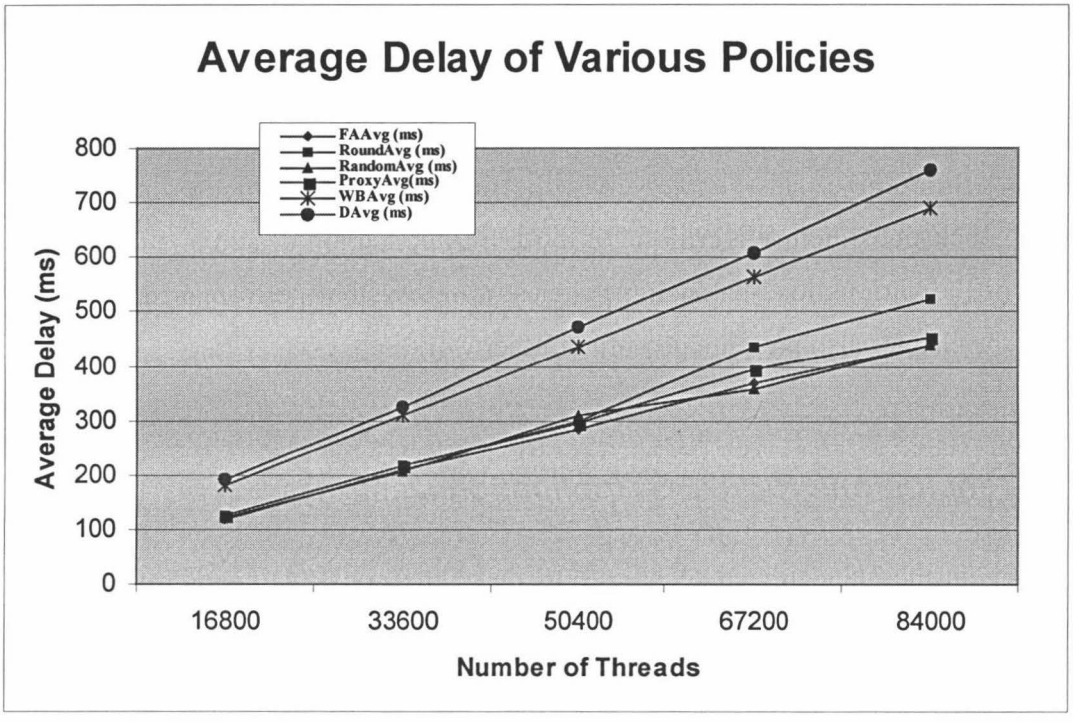


Figure 76: Average Delay of Various Policies

Chapter 8: Conclusions

Summary: This chapter gives the final conclusions of the thesis. The conclusions include a critique of the J2EE application design and the effect of the various load-balancing policies on the system performance. Finally, the chapter also presents potential further work.

8.1 Conclusions for J2EE Application Design

J2EE is the standardized technique to make the construction of n-tier distributed enterprise application easier. The technique tends to shift the programming burdens from the application developer to the container developer after EJB 2.1. All component design follows the standard patterns. This study constructed the benchmark J2EE application – *MGProject,* which shows a J2EE application is not complicated as it may seem if you follow some rules or steps.

- *Firstly, choose an appropriated IDE.* It is very important for constructing a J2EE application. A suitable IDE should have provided the framework of J2EE, including JDK and J2EE libraries, automatically generating deployment descriptors, providing the standard pattern for various components design etc. A suitable IDE can save lots of time and is less error prone. The NetBeans 4.1 or higher version is one of the ideal IDE for J2EE applications.
- *Carefully design the database.* Each table in the database represents an entity bean that can be stored permanently. After the database design is complete, the NetBeans IDE can automatically transfer it into coding of entity beans.
- *Analysis of the business processes.* The business processes are managed on session beans. At this level, whether the bean has to maintain client's session state should be decided. If it is, the bean should be a stateful bean. Otherwise, it is stateless. The EJB container manages the stateless beans in a more efficient manner than stateful beans.
- *Determining the layers and communication inside the application.* J2EE allows the application to have an n-tier layout as the *MGProject* used in this study or mix the several layers together such as direct communication between

servlet and database. It is recommended to have separate tiers in the J2EE application due to ease of maintainance.

- *Using EJB container to automatically manage bean persistence and relationships.* The EJB container can automatically manage CMP beans via deployment descriptors, which can be generated in NetBeans IDE directly. Nowadays, CMP with CMR is recommended for speed and is less error prone.
- *Select J2EE application server.* There are many J2EE vendors in today's market; each vendor requires specified deployment descriptors. The deployment descriptors are the key deployment documentation to make the J2EE application run smoothly. This research uses open source software JBoss to locate the *MGProject*.

On the whole, J2EE follows a standard way to construct the enterprise application plus the portable programming language Java, which allows the enterprise application to be "write once, deploy anywhere".

8.2 Conclusion for Load-Balancing Policies

The load-balancing policies play a very important role in the cluster environment. A good load-balancing policy can make the system optimize its function and present the best performance. The result brings to online customers the shortest response time and resulting in returning customers.

The study aims to find the most suitable load-balancing policy for the clustered application server JBoss. The experiments applied various load-balancing policies on the session bean level and examined the performance of the entire testing system via comparing the throughput of the system and the average time delay for waiting for HTTP responses. Six load-balancing policies where used in the testing, including four JBoss built-in policies, i.e. *FirstAvailable, FirstAvailableIdenticalAllProxies, RoundRobin* and *RandomRobin,* plus with two new created dynamic load-balancing policies, *dynamic* policy and *dynamic weight-based* policy. The experiments test these six policies on a balanced machine-load environment and on an unequal machine-load environment.

The experiments results indicate applying different load-balancing policies will have different outcomes. Particularly for the *RoundRobin* policy, the policy obviously becomes the bottleneck of the system on the unbalanced machine-load environment

although it has very good performance on a without any machine-load environment. The policy requires allocating almost equal amount of tasks among clustered members. If the same capability members have different workloads before accepting the new tasks, it should result in one of the members processing slowly and lead to decreasing the performance of the entire clustered system.

The experiments also presents the better performance of clustered system under any running environment can be obtained with the *FirstAvailable* and *RandomRobin* policies. And the *FirstAvailableIdenticalProxies* policy sometimes has good performance and sometimes poor. Comparing with the above four built-in policies, two dynamic policies present the worse performance in the balanced clustered environment. Such results are due to the dynamic policies generating the extra communications and calculations that give overloaded JBoss an extra burden. But the system with the dynamic weight-based policy has better performance than with the dynamic load-balancing policy. Since the dynamic weight-based policy optimizes every clustered server while the dynamic policy uses only one server during an interval.

Unlike built-in policies, the two dynamic policies present better balancing on unequal machine-load environment although the performance of entire system is low. The experiments prove the performance with dynamic policies is only a little bit below an unequal machine-load environment than on a balanced machine-load environment. With the increasing number of HTTP requests, the performance of system with dynamic policies are better and better via higher throughput and lower average time delay for HTTP responses. It is not the case for built-in policies; the load-balancing policies make the performance worse under unequal machine-load environment even becoming the bottleneck of the clustered system.

Although the load-balancing policies have been developed for a long time and it has been applied in various fields, dynamic load-balancing policies are not popular. The main reason it is the more intelligent policies usually require more complex control or complex calculations, which lead to the performance of the entire system dropping. It is the case in this study. On whole, the *FirstAvailable* and *RandomRobin* policies are more suitable load-balancing policies for clustered JBoss when the servers face heavy loads.

8.3 Further Work

A problem with the *MGProject* is primary key creation. For CMP bean, the container maintains the bean persistent relying on the primary key field. So in the *MGProject*,

each new instance of an entity bean will be assigned a unique key number before it is created. The *SeqidBean* provides the next primary key. However, when stress testing, a data-sharing problem exists. Some errors occur, such as *'cannot create entity since deadlock detected'*, *'entity with primary key ... already exists'*, *'next sequence block not available'* etc. All these problems occur since some threads have concurrent access to the same entity bean to get the next key value. JBoss provide one more property for defined *get** methods that are read-only. After *getSeqId* from the database, the field has been made dirty and has to be updated to avoid different session beans getting the same value, which cause duplicated key generation. Therefore, one more control for the *SeqidBean* is added via XDoclet (see figure 77) as for normal JBoss deployment descriptors.

```
@jboss.persistence   table-name="seqid"
                     check-dirty-after-get="true"
```

Figure 77: Adding check-dirty-after-get to JBoss Deployment Descriptors

However this time the NetBeans IDE cannot generate it in the deployment descriptors. Even adding it manually, the errors still occur on start up most of the time. Since each JBoss EJB container has a mechanism to control concurrent access, but for two or more JBoss, the concurrent access has to be controlled by the backend database not the EJB container itself.

In addition, even if a sequence block is introduced for each *seqSessionBean* and setting row-locking for *Seqid* table, it is still not enough to control the concurrent accessing problem. And when the JBoss faces heavy load, the container will increase the number of seqSessionBean instances automatically, which may cause *"possibly time out"* error if there are too many session beans waiting to fetch an ID block number. Meanwhile, before creating an instance of entity bean, it has to call on *seqSessionBean* and *SeqidBean* first, which also adds redundant steps reducing the performance of the application server. In fact, most modern databases have capability to automatically generate primary key if we defined the primary key field as auto-increment. Can we create CMP bean without primary key and leave it to underlying database as we directly insert a record to the database? This issue is still an argument among EJB developers. Some developers introduces "unknown-pk" declaration to deployment descriptor via XDoclet, such as

```
/**
*@jboss.entity-command   name="PostgreSQL-get-generated"
*@jboss.unknown-pk       class="java.lang.Integer"
*                        auto-increment="true"
*/
```

Figure 78: Try to Create Entity with unknown-pk for J2EE Application

Due to the study time limitation, the benchmark application used for testing the various load-balancing policies effect on the performance of system retains this problem. Such errors are not a critical issue for my testing purpose (there are only a few threads get errors). But as a J2EE developer, I think it is necessary to find the bug and fix it in the future.

Reference

[1] Zhou, M. Z. (2004). *A scalable application server on Beowulf cluster.*

[2] *Understanding Java and the J2EE platform.* Retrieved February 20, 2005, from http://media.wiley.com/product_data/excerpt/63/07645396/0764539663.pdf.

[3] Bodoff, S., Armstrong, E., Ball, J., Carson, D. B., Evans, I., Green, D., et al. (2004). *The J2EE tutorial.* (2nd ed.). Addison-Wesley.

[4] Ghaly, R. & Kothapalli, K. (2003). *Sams teach yourself EJB in 21days.* USA: Sams

[5] Kurniawan, B. (2002). *Java for the web with servlets, JSP and EJB.* Indiana: New Riders.

[6] Allamaraju, S., Browett, R., Diamond, J., Holden, M., Hoskinson, A., Johnson, Rl., et al. (2000). *Professional java server programming J2EE edition.* UK: Wrox Press Ltd.,

[7] *Java 2 platform enterprise edition specification, V1.4.* (2003). Retrieved May 6, 2005, from http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf

[8] Perry, B. W. (2004). *Java servlet and JSP cookbook.* Sebastopol: O'Reilly & Associates.

[9] Stearns, B. (2001). *Migrating from EJB 1.1 to 2.0.* Retrieved May 16, 2005, from http://java.sun.com/developer/technicalArticles/ebeans/ejbmigrate/

[10] *JBoss preferred by enterprise over IBM, Oracle, BEA.* Retrieved July 30, 2005, from http://java.sys-con.com/read/47746.htm?CFID=835383&CFTOKEN=C4A833ED-C3E0-AEB7-2A97D3A755155ABE

[11] *Apache Jmeter.* Retrieved August 16, 2005, from http://jakarta.apache.org/jmeter/index.html

[12] *Annotated postgresql.conf and Global User Configuration (GUC) Guide.* Retrieved May 26, 2005, from http://www.varlena.com/varlena/GeneralBits/Tidbits/annotated_conf_e.html

[13] Brookins, A. & Holloway, M. (Ed.) (2001). *Practical PostgreSQL.* Retrieved May 16, 2005, from http://www.commandprompt.com/ppbook/x293

[14] Monson-Haefel, R. (2002). *EJB 2.1: The timer services.* Retrieved September 19, 2005, from http://java.ittoolbox.com/documents/industry-articles/ejb-21-the-timer-

service-2084

[15] *Optimizing postgreSQL for N2H2 reporting.* (2004). Retrieved May 6, 2005, from http://www.n2h2.com/downloads/ifp_linux/v2.5/postgres_config.html

[16] *What is Mono?* Retrieved November 10, 2005, from http://www.mono-project.com/Main_Page

[17] *Servlet 2.4: what's in store.* Retrieved April 6, 2005, from http://www.javaworld.com/javaworld/jw-03-2003/jw-0328-servlet.html

[18] *Discover freely available servlet filters you can use today.* Retrieved May 3, 2005, from http://servlets.com/soapbox/filters.html

[19] *The essentials of filters.* Retrieved April 6, 2005, from http://java.sun.com/products/servlet/Filters.html

[20] *Local and remote EJB interfaces.* Retrieved March 13, 2005, from http://www.onjava.com/pub/a/onjava/2004/11/03/localremote.html

[21] Deshpande, S. & Martin, B. (2001). *Eight reasons ECperf is the right way to evaluate J2EE performance.* Retrieved April 20, 2005, from http://www.theserverside.com/articles/article.tss?l=Why_ECperf

[22] Zhang,Y., Liu, A. & Qu, W. (2003). *Comparing industry benchmarks for J2EE application server: IBM's Trade2 vs Sun's Ecperf.* ACM International Conference Proceeding Series. Vol. 35.

[23] *SPEC release new application server benchmark.* Retrieved May 24, 2005, from http://www.spec.org/jAppServer2004/press_release.html

[24] White, A. *Measuring the benefits of Ajax.* Retrieved October 20, 2005, from http://www.developer.com/java/other/article.php/10936_3554271_1

[25] *Distributed computing.* Retrieved July 20, 2005, from http://en.wikipedia.org/wiki/Distributed_computing

[26] Minar, N. (2002). Distributed systems topologies. Retrieved August 11, 2005, from http://www.openp2p.com/pub/a/p2p/2002/01/08/p2p_topologies_pt2.html

[27] Tanenbaum, A. S. & Steen, M. V. (2002). *Distributed system principles and paradigms.* New Jersey: Prentice-Hall.

[28] Yu, W. (2005). *Uncover the hood of J2EE clustering.* Retrieved September 15, 2005, from http://www.theserverside.com/articles/article.tss?l=J2EEClustering

[29] *Beowulf (computing).* Retrieved May 10, 2005, from

http://en.wikipedia.org/wiki/Beowulf_(computing)

[30] *What makes a cluster a Beowulf?* Retrieved May 20, 2005, from http://www.beowulf.org/overview/index.html

[31] *A first look at Linux clustering.* (2001). Retrieved August 2, 2005, from http://linux.omnipotent.net/article.php?article_id=12019

[32] Penchikala, S. (2004). *Clustering and load balancing in Tomcat 5, part 1.* Retrieved October 26, 2005, from http://www.onjava.com/lpt/a/4649

[33] Kang, A. (2001*). J2EE clustering, part 2 – migrate your application from a single machine to a cluster, the easy way.* Retrieved July 20, 2005, from http://www.javaworld.com/javaworld/jw-08-2001/jw-0803-extremescale2.html

[34] *Using NetBeans with the JBoss getting started guide.* Retrieved March 12, 2005, from http://www.netbeans.org/kb/50/jboss-getting-started.html

[35] *NetBeans 4.1 tutorials, guides and articles.* Retrieved February 20, 2005, from http://www.netbeans.org/kb/41/

[36] Millson, M. (2003). *Integrating Tomcat and Apache on RedHat 9.0.*

[37] *Boost for supercomputing in New Zealand.* Retrieved November 20, 2005, from http://www.supercomputingonline.com/article.php?sid=7512

[38] *Helix supercomputer set to double.* Retrieved November 20, 2005, from http://masseynews.massey.ac.nz/2004/Press_Releases/28_24_04.html

[39] *Double Helix.* Retrieved November 22, 2005, from http://double-helix.massey.ac.nz/

[40] *The Allan Wilson Center & Massey University parallel processing cluster.* Retrieved September 20, 2005, from http://helix.massey.ac.nz/

[41] *The trusted leader in middleware software.* Retrieved May 5, 2005, from http://www.jboss.com/

[42] *Getting started with JBoss 4.0.* (2004). Retrieved February 28, 2005, from http://docs.jboss.org/jbossas/getting_started/v4/html/

[43] *The JBoss 4 application server guide.* (2005). Retrieved March 26, 2005, from http://docs.jboss.org/jbossas/jboss4guide/r4/html/

[44] Labourey, S. & Burke, B. (2004*). JBoss AS clustering.* (7th ed.). Retrieved February 20, 2005, from http://docs.jboss.org/jbossas/clustering/JBossClustering7.pdf

[45] Babaoğlu, Ő., Batoli, A., Maverick, V., Montresor, A., Rossi, D. & Vučković, J. (2003). *JBoss clustering analysis.*

[46] *Apache Tomcat user guide.* Retrieved April 2, 2005, http://jakarta.apache.org/jmeter/index.html

[47] Leonard, B. (2005). *Integrating NetBeans with other J2EE server vendors.* Retrieved March 20, 2005, from http://www.netbeans.org/kb/41/j2ee-server-integration.html

[48] Panda, D. (2004). *Automatically scheduling a timer.* Retrieved September 15, 2005, from http://www.onjava.com/lpt/a/5267

[49] Wang, B. (2005). *JBoss cache as a POJO cache.* Retrieved October 2, 2005, from http://www.onjava.com/pub/a/onjava/2005/11/09/jboss-pojo-cache.html

[50] Ban, B., Wang, B., Surtani, M., & Stansberry, B. *TreeCache: a tree structured replicated transactional cache*. Retrieved October 20, 2005, from http://docs.jboss.org/jbcache/1.2.4/TreeCache/html/

[51] Kung, C. (2005). *Jmeter tips – improve the quality of your Jmeter scripts*. Retrieved September 13, 2005, from http://www.javaworld.com/javaworld/jw-07-2005/jw-0711-jmeter.html

[52] *Middleware – software technology roadmap*. Retrieved May 18, 2005, from http://www.sei.cmu.edu/str/descriptions/middleware_body.html

[53] Chacko, V. (2005). *Multiversion concurrency control (MVCC) – how PostgreSQL attain data consistency*. Retrieved October 12, 2005, from http://searchwarp.com/swa9860.htm

[54] Sun Microsystems. (2002). *Ecperf™ kit version 1.1 – final release*. Retrieved March 12, 2005, from http://java.sun.com/developer/earlyAccess/j2ee/ecperf/download.html

[55] *Client/Server and the N-Tier Model of distributed computing*. Retrieved September 2, 2005, from http://n-tier.com/articles/csovervw.html

[56] Sun Microsystems. (2003). *Enterprise JavaBeansTM Specification, version 2.1.* Retrieved March 10, 2005, from https://sdlc5c.sun.com/ECom/EComActionServlet;jsessionid=C961B10E7E3044277CEFE81220B84E66

Appendix

In this part, I explore the step-by-step operations including installation and configuration of the open source software I used for my study. Please note: all software I installed on a Linux environment. So at most cases, we need to give permission to run the software firstly, using command:

> ***chmod a+x software_name***

A. *IDE – NetBeans*

1. Installation

The NetBeans community provides standalone IDE installation and bundled with Sun Java Application Server Platform Edition 8. We develop a J2EE application we need an application server. So I downloaded NetBeans IDE with SUN AS 8 from http://www.netbeans.org/downloads/index.html. Before you install the IDE, you must ensure your machine has installed JDK 1.4 or higher, and provides the classpath to it. The installation command is as following:

> sjsas_pe-8_1_02_2005Q2-nb-4_1-linux.bin is:javahome /home/xchen/j2sdk1.4.2_08

B. *PostgreSQL*

1. Installation

The requirement for installation PostgreSQL is simple, only an Unix-like system that builds with GNU tools. The installation package can be downloaded from PostgreSQL website http://www.postgresql.org/ftp/source/v7.4.9/. The chapter of installing PostgreSQL [13] gives "10 steps to installing PostgreSQL" in detail. But the instructions assume you have system administrator privilege. In my study, the database is only created for my study use, it is directly located in my home directory. So some steps can be ignored.

1. Uncompress *postgresql-<version> .tar.gz* installation package
2. Go into the postgresql home directory, then configure the source tree: ***./configure –with-java***
3. Compile the source: ***gmake*** or ***make***
4. Regression testing: ***gmake (make) check***
5. Install: ***gmake (make) install***
6. Initializing the database***: bin>initdb –D ../data***
7. Start PostgreSQL: ***bin>pg_ctl –o "-i" –D ../data start***

(The option "-i" enables the database server with TCP/IP connection)

2. Database in PostgreSQL

To create the *mgDB* in PostgreSQL is not too complex. Firstly, we need to start the database PostgreSQL. The next step we can create a database as used in this study: ***bin>createdb mgDB.*** And then follow the definition of PostgreSQL to create tables.

To access the database, we can load the definition of tables to database *mgDB* manually or from file use command:

bin> psql mgDB;

(then paste mgDB.sql content here by manual)

or

bin> psql –f /home/xchen/MasterThesis/mg.sql mgDB xchen (load from file)

3. Connection Configuration in PostgreSQL

The DataSource connection to PostgreSQL uses the JNDI name "*java:/PostgresDS*" in JBoss. For the steps of basic configuration of JBoss with PostgreSQL see Appendix C.3. The XML file *postgres-ds.xml* under the JBoss deploy directory is the JDBC connection file that will be read when JBoss starts, which states the JDBC connection port and the name or IP address of the database server, and the name of the database, the user name and password to access the database. The default PostgreSQL connection port number is 5432.

As stated before, PostgreSQL is sensitive to the size of the shared buffer since it uses shared memory for carrying out its work. When JBoss starts, it automatically establishes the connection to the PostgreSQL with the minimum connection value defined in *postgres-ds.xml*. If there is not a minimum connections defined in *–ds.xml* file, the default value minimum connections 5 and maximum connections of 20 will be used by JBoss. The number of connections and allocated share memory for the PostgreSQL can be checked from the Linux command in the PostgreSQL machine:

```
ipcs -m
```

The default connections value always runs out of shared memory. The default value of shared buffer size 1024 bytes is increased to 4028 bytes in my case. The modified *postgres-ds.xml* with connections definition is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
    <local-tx-datasource>
            <jndi-name>PostgresDS</jndi-name>
            <connection-url>jdbc:postgresql://sisters:5432/mgDB</connection-url>
            <driver-class>org.postgresql.Driver</driver-class>
            <user-name>xchen</user-name>
            <password></password>
            <min-connections>50</min-connections>
            <max-connections>200</max-connections>
            <metadata>
                <type-mapping>PostgreSQL</type-mapping>
            </metadata>
    </local-tx-datasource>
</datasources>
```

C. *JBoss*

1. JBoss installation

Make sure the system has Java Development Kit (JDK) 1.4 or higher to install, not just Java Runtime Environment (JRE). And make sure the JAVA_HOME points to the JDK home directory. The latest version of JBoss is v 4.0.3. The compressed version of JBoss can be downloaded from JBoss web site. After unzipping the file, you access the server directory under the JBOSS_HOME directory, you can find there are three preconfigured server types: *all, default, minimal*. Only the *all* configuration can support the cluster feature. So start up the JBoss:

JBOSS_HOME/bin> ***run –c all***

If you have a successful installation, you can view JBoss Management Console by clicking http://localhost:8080/jmx-console . The web page displays the JMX MBeans that make up the server.

2. Configuring JBoss with PostgreSQL

The JBoss AS has a flexible mechanism to accept other data resources rather than using the default database Hypersonic. To use JBoss AS with database PostgreSQL with JDBC connection, we need:

Step 1: Add driver class to classpath: We copy the driver class postgresql.jar to the server library. For example, in my case I need the cluster characteristic, so I copy this driver class under *<jbossHome>/server/all/lib* directory.

Step 2: Configuring the PostgreSQL datasource: JBoss has predefined some popular datasource files under *<jbossHome>docs/examples/jca/* directory. Here, we copy *postgres-ds.xml* to the *<jbossHome>/server/all/deploy* directory and modify some parameters to meet our needs. This file will be deployed when the JBoss server is started. The following is the *postgres-ds.xml* for my application.

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
    <local-tx-datasource>
        <jndi-name>PostgresDS</jndi-name>
        <connection-url>jdbc:postgresql://sisters:5432/mgDB</connection-url>
        <driver-class>org.postgresql.Driver</driver-class>
        <user-name>xchen</user-name>
        <password></password>
            <metadata>
            <type-mapping>PostgreSQL</type-mapping>
            </metadata>
    </local-tx-datasource>
</datasources>
```

Step 3: Configuring some files under <jbossHome>/server/all/conf/ directory with PostgreSQL setting.

- Modifying *standardjaws.xml* file and adding:

```
<jaws>
    <datasource>java:/PostgresDS</datasource>
    <type-mapping>PostgreSQL</type-mapping>
    ...
</jaws>
```

- Modifying *standjbosscmp-jdbc.xml* file:

```
<defaults>
    <datasource>java:/PostgresDS</datasource>
    <type-mapping>PostgreSQL</type-mapping>
    ...
</defaults>
```

- Modifying *login-config.xml* and adding:

```
<application-policy name = "PostgresDbRealm">
    <authentication>
        <login-module code = "org.jboss.resource.security.ConfiguredIdentityLoginModule"
            flag = "required">
            <module-option name = "principal">sa</module-option>
            <module-option name = "userName">xchen</module-option>
            <module-option name = "password"></module-option>
            <module-option name = "managedConnectionFactoryName">
                    jboss.jca:service=LocalTxCM,name=PostgresDS</module-option>
        </login-module>
    </authentication>
</application-policy>
```

Step 4: copy <jbossHome>/docs/examples/jms/postgres-jdbc2-service.xml to <jbossHome>/server/all/deploy-hasingleton/jms/ and remove hsql-jdbc2-service.xml. After that, rename hsql-jdbc-state-service.xml to postgres-jdbc-state-service.xml and modify the file content from "DefaultDS" to "PostgresDS". Which enables the JBoss state management using PostgresDS.

D. *Apache & Tomcat*

1. Download Files

To install Tomcat as clustered web servers, we need Tomcat, Tomcat connector and Apache HTTP Server. They can be downloaded from the following location.

- Download Apache Tomcat 5.5.9 version from http://archive.apache.org/dist/tomcat/tomcat-5/archive/ . Then uncompress the archive *.tar.gz* file in Linux environment.
- Download Apache HTTP Server 2.0.55 from http://httpd.apache.org/download.cgi. Uncompress the archive file.
- Download Tomcat connector JK2 source archive file from http://tomcat.apache.org/download-connectors.cgi.

2. Install Apache

To build mod_jk2 from the source version, we need the *apxs* tool, which is available after install of Apache HTTP Server. The installation of Apache follows the 3 steps:

- To go into the httpd-2.0.55 directory, then

  ./configure --prefix=/home/xchen/apache20

  the above *apache20* will be an installation directory after configuring the system.

- To compile the apache HTTP server, use the command: ***make***
- The last installation command: ***make install*** will copy the files into the predefined directory *apache20.*

After installing the apache, you can get the *apxs* tool under the *apache20/bin* directory.

3. Install Tomcat connector mod_jk2

When the *apxs* tool is ready, you can get into the *<tomcat-connector>/jk/native2* and follow the instructions in BUILD.txt. That is:

./configure –with-apxs2=/home/xchen/apache20/bin/apxs

make

cd ../build/jk2/apache2

/home/xchen/apache20/bin/apxs –n jk2 –i mod_jk2.so

We notice the connector *mod_jk2* is built inside the Apache HTTP server. Because of installing *mod_jk2* using Apache *apxs* tool, it will directly generate and copy the *mod_jk2.so* file into the Apache server under the *modules* directory. The *mod_jk2* uses a worker (i.e. instance of Tomcat server) concept to establish the communication between Apache server and Tomcat server. Therefore, the configuration for Tomcat clustering is simpler. A *worker2.conf* file under the */conf/* in the Apache is created, which specify the host and connection port number of each clustering Tomcat instance, and load-balancing policy. In addition, each Tomcat instance also needs to present a different connection port number in *server.xml* file if they are running on the same machine. The connection port number should identify the numbers present in the *worker2.conf* file in Apache.

4. Configuring Tomcat 5.5.9

For tomcat to run in Linux environment, you must have a suitable JDK environment. I installed JRE 1.5 for it. There is no extra configuration after Tomcat installing. But you have to provide the library files for the web application either on <tomcatHome>/common/lib or <tomcatHome>/shared/lib. In my case, the web application archive file is ready. So the file MGProject-WebModule.war file just directly drops into the servlet container --- <tomcatHome>/webapps/. The tomcat will automatically expand the .war file into a folder. I also put library files *jbossha.jar, jbossallclient.jar* and *jbossj2ee.jar* under the *<tomcatHome>/shared/lib* that are need by my web application.

E. *Jmeter*

First of all, make sure you have installed JDK, for the most recent Java-based application, it requires JDK1.4 or higher. Then download Jmeter from http://jakarta.apache.org/site/downloads/downloads_jmeter.cgi. Unzip the binary file into a directory, i.e. *Jmeter_Home*. Changing to *bin* directory and executes the following command:

Jmeter_Home/bin>jmeter

The command will bring up the Jmeter graphic user interface. From the GUI, you can configure stress tests via creating a Test Plan.

F. *Database - mg.sql*

```
--Store discount rule and percentage

DROP TABLE discount Cascade;
CREATE TABLE discount (
  d_type text UNIQUE NOT NULL,
  d_rule text NOT NULL,
  d_percent int,
  CONSTRAINT discount_d_type_pkey PRIMARY KEY (d_type)
);

INSERT INTO discount VALUES('Gold Customer','A customer prints material with total value above
$500,000',25);

INSERT INTO discount VALUES('Silver Customer','A customer prints material with total value above
$300,000',18);

INSERT INTO discount VALUES('Longterm Customer','A customer prints material with total value
above $100,000',10);

INSERT INTO discount VALUES('First Time Customer','A customer prints material first time',5);

INSERT INTO discount VALUES('General Customer','Just a customer',0);

SELECT * FROM discount;


--Record customer information
--DROP SEQUENCE customer_c_id_seq;
DROP TABLE customer CASCADE;
CREATE TABLE customer (
  c_id serial UNIQUE NOT NULL,
  c_companyName text,
  c_contactName text NOT NULL,
  c_address text,
  c_phone text NOT NULL,
  c_email text,
  c_fax text,
  c_since date NOT NULL,
  c_totalAmount numeric(15,2),
  c_d_type varchar(50),
  CONSTRAINT customer_c_id_pkey PRIMARY KEY (c_id),
  FOREIGN KEY(c_d_type) REFERENCES discount (d_type)
);

INSERT INTO customer VALUES (nextval('customer_c_id_seq'),null,'xiao
wu',null,'8562356','xiaowu@hotmail.com',null,CURRENT_DATE,1234.75,'First Time Customer');

INSERT INTO customer VALUES (nextval('customer_c_id_seq'),'Sun Explore Ltd.','jackie lim','254
bush rd., Albany, Auckland','4785896','jackielim@sunex.co.nz','4785898','4/12/2002',560786.50,'Gold
Customer');

INSERT INTO customer VALUES (nextval('customer_c_id_seq'),null,'mary lee','56 lake st, Three King,
Auckland','6972566','m.lee@xtra.co.nz',null,'5/25/2000',323652.95,'Silver Customer');

INSERT INTO customer VALUES (nextval('customer_c_id_seq'),null,'lui','156 black st, Three King,
Auckland','6589963','m.lui@hotmail.com',null,'5/28/2003',323653.95,'Silver Customer');


SELECT * FROM customer;
```

```
--Store parts detail used by corp

DROP TABLE parts CASCADE;
CREATE TABLE parts (
  p_id integer UNIQUE NOT NULL,
  p_name text NOT NULL,
  p_desc text NOT NULL,
  p_code text,
  p_unit text NOT NULL,
  p_cost numeric(10,2),
  p_price numeric(10,2),
  p_lomark integer,
  p_himark integer,
  CONSTRAINT parts_p_id_pkey PRIMARY KEY (p_id)
);

INSERT INTO parts VALUES (1, 'paper', 'economic paper 60g',null,'sheet',0.60,0.70,50,300);
INSERT INTO parts VALUES (2, 'paper', 'economic paper 80g',null,'sheet',0.75,0.83,50,300);
INSERT INTO parts VALUES (3, 'paper', 'gloss paper 128g',null,'sheet',1.00,1.10,50,300);
INSERT INTO parts VALUES (4, 'paper', 'gloss premium 157g',null,'sheet',1.21,1.35,30,100);
INSERT INTO parts VALUES (6, 'ink', 'red','r36','tin',15.5,null,5,15);
INSERT INTO parts VALUES (7, 'ink', 'blue','b23','tin',14.5,null,5,15);
INSERT INTO parts VALUES (8, 'ink', 'green','g13','tin',15.25,null,5,15);
INSERT INTO parts VALUES (9, 'petrol', 'petrol',null,'liter',3.5,null,5,15);
INSERT INTO parts VALUES (10, 'cloth', 'old cloth',null,'bundle',0.9,null,20,100);
INSERT INTO parts VALUES (11, 'ps', 'ps piece','ps19','piece',35.00,50.00,20,200);
INSERT INTO parts VALUES (12, 'office paper', 'A4','Reflex','pack',5.00,10,30);
INSERT INTO parts VALUES (13, 'paper', 'gloss lite paper 80g',null,'sheet',0.75,0.83,50,300);

SELECT * FROM parts;


--Store all parts in inventory

DROP TABLE inventory CASCADE;
CREATE TABLE inventory (
  in_id integer UNIQUE NOT NULL,
  in_quantity integer NOT NULL,
  in_ordered integer,
  in_location text,
  in_activeDate date,
  CONSTRAINT intentory_in_id_pkey PRIMARY KEY (in_id),
  FOREIGN KEY(in_id) REFERENCES parts (p_id)
);

INSERT INTO inventory VALUES (1, 120, -30, 'manufactory', '4/6/2005');
INSERT INTO inventory VALUES (2, 60,+50, 'manufactory', '5/10/2005');
INSERT INTO inventory VALUES (3, 80, -40, 'manufactory','5/12/2005');
INSERT INTO inventory VALUES (4, 60,+20,'manufactory', '4/15/2005');
INSERT INTO inventory VALUES (6, 10, -2, 'manufactory', '6/12/2005');
INSERT INTO inventory VALUES (7, 15,-1,'manufactory','6/18/2005');
INSERT INTO inventory VALUES (8, 10, +5, 'manufactory', '3/15/2005');
INSERT INTO inventory VALUES (9, 10, -3, 'manufactory', '11/23/2004');
INSERT INTO inventory VALUES (10, 50, -5, 'manufactory', '6/7/2005');
INSERT INTO inventory VALUES (11, 45, -5, 'manufactory', '5/25/2005');
INSERT INTO inventory VALUES (12, 20,-1,'office', '7/8/2005');

SELECT * FROM inventory;


DROP TABLE orders CASCADE;
```

```
CREATE TABLE orders (
  o_id serial NOT NULL,
  o_c_id integer,
  o_ol_count integer,
  o_discount numeric(4,2),
  o_total numeric(12,2),
  o_status text,
  o_entryDate date,
  o_shipDate date,
  CONSTRAINT orders_o_id_pkey PRIMARY KEY (o_id),
  FOREIGN KEY(o_c_id) REFERENCES customer (c_id)
);

INSERT INTO orders VALUES (nextval('orders_o_id_seq'),2,1, 0.25, 2356.5, 'shipped', '11/25/2004',
'1/28/2005');
INSERT INTO orders VALUES (nextval('orders_o_id_seq'),1,2, 0.05, 986.75, 'outstanding', '6/25/2005',
null);
INSERT INTO orders VALUES (nextval('orders_o_id_seq'),3,1, 0.15, 1986.75, 'outstanding', '6/12/2005',
null);
INSERT INTO orders VALUES (nextval('orders_o_id_seq'),3,1, 0, 986.75, 'processing', '7/12/2005', null);

SELECT * FROM orders;




--The order detail
--DROP SEQUENCE item_i_id_seq;
DROP TABLE item CASCADE;
CREATE TABLE item (
  i_id serial NOT NULL,
  i_name text NOT NULL,
  i_size text,
  i_paperDesc text,
  i_color integer,
  i_colorDesc text,
  i_doubleSide boolean,
  i_innerPaperDesc text,
  i_innerPageColor integer,
  i_innerPageColorDesc text,
  i_totalPageNo integer,
  i_requirement text,
  CONSTRAINT item_i_id_pkey PRIMARY KEY (i_id)
);

INSERT INTO item VALUES (nextval('item_i_id_seq'),'book', '256 * 190', 'gloss premium paper 157g',
4, 'colored', '0', 'economic paper 60g', 1, 'black', 354, null);
INSERT INTO item VALUES (nextval('item_i_id_seq'),'advertisement sheets', '368 * 240', 'gloss paper
128g', 2, 'lite green background, blue text', '0', null,null,null, null,null);
INSERT INTO item VALUES (nextval('item_i_id_seq'),'advertisement sheets', '312 * 220', 'gloss
premium paper 128g', 4, 'colored', '0', null,null,null,null, null);
INSERT INTO item VALUES (nextval('item_i_id_seq'),'advertisement', '256 * 190', 'gloss premium
paper 157g', 4, 'colored', '1', null,null,null,null, null);

SELECT * FROM item;



DROP TABLE orderline CASCADE;
CREATE TABLE orderline (
  ol_id serial NOT NULL,
  ol_o_id integer,
```

```
  ol_i_id integer,
  ol_quantity integer,
  ol_sumPrice numeric(10,2),
  ol_status text,
  ol_shipDate date,
  CONSTRAINT orderline_ol_id_pkey PRIMARY KEY (ol_id),
  FOREIGN KEY(ol_o_id) REFERENCES orders (o_id),
  FOREIGN KEY(ol_i_id) REFERENCES item (i_id)
);

INSERT INTO orderline VALUES (nextval('orderline_ol_id_seq'), 1, 1, 250, 2356.5, 'shipped',
'1/28/2005');
INSERT INTO orderline VALUES (nextval('orderline_ol_id_seq'), 2, 3, 1000, 356.5, 'outstanding', null);
INSERT INTO orderline VALUES (nextval('orderline_ol_id_seq'), 2, 2, 2000, 630.25, 'outstanding', null);
INSERT INTO orderline VALUES (nextval('orderline_ol_id_seq'), 3, 4, 3000, 1986.75, 'outstanding',
null);

SELECT * FROM orderline;

DROP TABLE seqID CASCADE;
CREATE TABLE seqID (
  s_id text not null,
  s_nextNum int,
  s_blocksize int,
CONSTRAINT seqID_s_id_pkey PRIMARY KEY (s_id)
);

INSERT INTO seqID VALUES ('customer',5,100);
INSERT INTO seqID VALUES ('item',5,100);
INSERT INTO seqID VALUES ('orders',5,100);
INSERT INTO seqID VALUES ('orderline',5,100);

select * from seqID;


DROP TABLE loadAvg CASCADE;
CREATE TABLE loadAvg (
  hostname text UNIQUE NOT NULL,
  firstAvg numeric(5,2),
  secondAvg numeric(5,2),
  thirdAvg numeric(5,2),
  ratio int,
  countRatio int,
  CONSTRAINT loadAvg_hostname_pkey PRIMARY KEY (hostname)
);
```

G. *Sample Bash File*

The bash file configures to test the performance of clustered two JBoss. The following is a sample of bash file, which automatically start applications to perform tests.

```
#kill all applications
killTwo
sleep 5

#start JBoss
ssh amd7 /mnt/scratch/xchen/jboss-4.0.3SP1/bin/freshmg &
sleep 35
ssh amd8 /mnt/scratch/xchen/jboss-4.0.3SP1/bin/freshmg &
#ssh amd1 /mnt/scratch/xchen/jboss-4.0.3SP1/bin/freshmg &

sleep 10

#start two Tomcat
ssh amd3 /home/xchen/tomcat1/bin/freshmg
ssh amd4 /home/xchen/tomcat2/bin/freshmg

sleep 20
#start Jmeter in non-distributed mode

ssh amd5 /home/xchen/jakarta-jmeter-2.0.3/bin/jmeter -t ~/twoTomcatNoTimer.jmx

#use for distributed Jmeter
#ssh amd1 /home/xchen/jakarta-jmeter-2.0.3/bin/jmeter-server &

#ssh amd5 /home/xchen/jakarta-jmeter-2.0.3/bin/jmeter-server &

#ssh amd5 ~/jakarta-jmeter-2.0.3/bin/jmeter -t ~/twoTomcatNoTimer.jmx -
#remote_hosts=amd5,amd1
```