UNIVERSITY OF OSLO
Department of Informatics

Analysis of a Java Enterprise Application stack

Master thesis

Petter Asskildt

22.05.2007

Table of contents

Table of Figures

1 Introduction

Client-server architecture is a well documented and popular approach to design of distributed systems. A server that provides a set of services (applications) to various clients is called an application server; such servers have a common set of needs to fulfil their role. These needs are addressed by providing a software package that encapsulates many of these areas of functionality into one bundle. Many different software bundles exist, but probably the most renowned ones are Red Hat's *JBoss*, IBM's *WebSphere*, Oracle's *Oracle Application Server* and *BEA WebLogic.*

Almost every enterprise application is required to persist various data beyond the execution time frame, being customer information, product information or whatever data is needed to provide certain functionality over time. A typical application server consists of an application server frontend which hosts a number of enterprise applications that does all communication with the clients and a backend database which ensures persistent data. The work of mapping relational data (relational databases store data in relations, much different from the object oriented approach commonly used in applications today) to objects within the enterprise applications running on the application server is tedious and error prone, this mapping is provided by the application server software bundle to ease development of enterprise applications.

Sun provides JavaEE (Java Enterprise Edition version 5) as a framework for developing server side applications. One of the major parts of Java Enterprise Edition is Enterprise Java Beans (EJB) specification. EJB is a framework aiding developers in creating frontend interfaces for the clients (called session beans) and classes representing persistent data (called entity beans). These classes are *plain old java objects* (POJO) with annotations that define how the application server should handle these classes (more on this in sections 2.2 and 2.3).

Web services is a new communication standard between server and client and is based on XML as data representation language and WSDL[1] as definition language. The server publishes via WSDL what services it can provide and the client can connect to the web service endpoint to utilize this service. Web services have gained broad acceptance as the new standard for language neutral software communication.

This thesis will focus on a detailed analysis of a Java EE software stack with primary focus on the persistence and object relational mapping section. To achieve this, a system that mimics the TPC-App (Transaction Processing Performance Council, 2005) specification will be used (see section 3.1 for further details).

[1] WSDL: Web Service Definition Language

2 Technologies and middleware involved

2.1 Application Server

An application server is a computer server that is dedicated to running applications in contrast to other server types like file servers or printer servers. Application servers are typically used in environment where some common business logic is used by numerous clients. This business logic often relies on data access which when executed on a single server can be maintained by the server in contrast to systems where no single entity is responsible as a whole, but all participants in the distributed system work together to obtain a common service (e.g. the Freenet Project, a distributed anonymous information storage and retrieval system). In situations where one part of the system requires knowledge of another (e.g. what time did an event occur) or is dependent on that other part running smoothly using a distributed model with no central control unit is very complex (e.g. *global clock* and *independent failure* problems that occur in distributed systems (Coulouris, Dollimore, & Kindberg, 2005)).

Application servers typically include a bundle of middleware to ease the programming effort needed to develop a client-server application (e.g. security, transactions …). Sun's Java Enterprise Edition is a very popular framework for developing software designed for client-server environment, and with the release of version 1.5 (known as JavaEE) Sun introduced the Enterprise Java Beans version 3.0 which dramatically reduces the effort needed to develop an enterprise application (see section 2.2 below)

Enterprise JavaBeans (EJB) is a server-side component architecture that is embedded in most java based application servers. "Server-side component models are used on middle-tier application servers, which manage the components at runtime and make them available to remote clients. They provide a baseline of functionality that makes it easy to develop distributed business objects and assemble them into business solutions" (Burke & Monson-Haefel, 2006)

2.1.1 JBOSS

JBoss is an open source application server software developed by Red Hat. This software package is popular among small to medium businesses and private use due to the fact that it is open source and free. Other non open source application servers like IBM's WebSphere are costly to set up, but larger enterprises often cannot rely on an open source community when support is needed. A new type of open source solutions are gaining leverage, called *professional open source software*, these solutions are backed by companies and not only communities of volunteers and provide extensive support programs along with their products; JBoss is such a software package.

2.2 Enterprise Java Beans (EJB)

The Enterprise JavaBeans architecture is component architecture for the development and deployment of component- based distributed business applications. According to Sun "EJB technology enables rapid and simplified development of distributed, transactional, secure and portable applications based on Java technology" (Sun, 2007)

EJB is designed to help build applications for enterprise use. The key difference between a regular application installed on a local machine (i.e. Microsoft Word) and an enterprise application (component) is that the enterprise application is running on a server and clients connects to this central application. This means that a typical enterprise application is installed on some server and provides a set of services to a number of clients. There are several issues that need to be addressed in such an environment. To name a few, the enterprise application must handle a large number of concurrent clients, concurrent data access and security between server and client. These issues are not easily developed in a single project as it is often very complex. EJB addresses these problems and provides developers with these functionalities automatically.

Recently the new 3.0 version of Java Enterprise Beans was released; this is a major upgrade and redefinition of the old 2.1 version. This new version of the architecture focuses primarily on making the technology easier to use by developers (Ort, 2004), this, because version 2.1 was seen by many developers as too complex and tedious to use.

2.3 Object Relational Mappings (ORM)

Object relational mapping is the task of making data persistent, typically objects managed by a program. One way of doing this is simple serialization, but this is not feasible when handling large amounts of objects. The solution is to use traditional relational databases which have sophisticated features for handling large amount of data and also concurrent access to these data.

However, databases use relations to store data and have no concept of an object as an entity or encapsulation. Relational databases store data rows (called records) in tables. Hence one has to perform some sort of conversion from the class structure used in object oriented programming to the relational structure in relational databases. This has traditionally been done by mapping the attributes of an object to columns in the data table. Each row (record) in the table represents one object. This may sound like a straight forward approach, but there are many pitfalls and writing safe code for this mapping is tedious work.

Because of the general need for persisting data some tools have been developed to aid the developer in this mapping between object structures to relational structures. These tools maintain the ACID[2] properties required in database processing.

Most ORMs introduce an object oriented query language which supports the well known “dot” notation of accessing fields and members of objects. This significantly reduces the complexity of the queries performed by the programmer. The translation from this object oriented language to the standard SQL query is done by the ORM, making it transparent to the developer.

[2] ACID: stands for Atomicity, Consistency, Isolation, and Durability. They are considered to be the key transaction processing features/properties of a database management system (Navathe, 2004)

2.3.1 Hibernate

Hibernate is an open source object relation mapping tool. Hibernate supports all the features described above. It is developed to support the Java programming language, but Hibernate also develops an open source version for .NET called NHibernate. Hibernate is the only ORM that supports both Java and .NET through the same API.

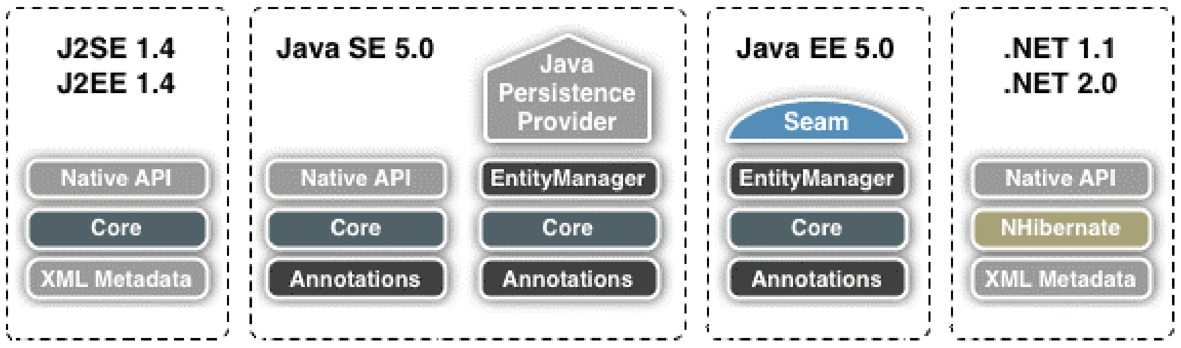

Figure 1 Hibernate Stack (Hibernate.org, 2007)

As can be seen from the Hibernate stack above, Hibernate supports annotations introduced in Java version 5.0 through the Java Persistence API (JPA). This enables developers to set up the in-code persistence mapping eliminating the need for additional mapping files. One may argue that this introduce complexity when it comes to changing the application persistence configuration (e.g. port the application to another database). To avoid this Hibernate supports both mapping files and in-code mapping annotations. The mapping files supersede the in-code annotations giving developers the possibility of setting up a default configuration in code, and subsequently override this setup with mapping files. This overriding external mapping is not required to be exhaustive, which is particularly convenient as one can change only the desired parameters (and one does not need to know the entire mapping configuration, although a correct total mapping is of course required).

2.4 Java Persistence API (JPA)

The Java Persistence API is a POJO[3] persistence API for object relational mapping. It contains a full object/relational mapping specification supporting the use of Java language metadata annotations and/or XML descriptors to define the mapping between Java objects and a relational database. It supports a rich, SQL-like query language (which is a significant extension upon EJB QL) for both static and dynamic queries. It also supports the use of pluggable persistence providers. (Sun, 2007)

External descriptors are often used in conjunction with annotations. The external descriptor will override annotations written in code (if the descriptor specifies some value that is already specified by annotations). This way you have a default setup in the annotations, and the possibility to tailor this setup with an external descriptor, typically done when the application is ported to a different backend database.

The Java Persistence API contains a collection of predefined annotations that are used to describe persistent fields, beans, etc. E.g. the @Column annotation specifies that a field within a class should be mapped to a column in a specific table.

[3] POJO: Plain Old Java Objects

JPA is Sun’s effort to standardise object relational mapping, Hibernate, TopLink and many other ORMs have existed for some time, and with success; hence Sun has now built a standard for object relation mapping drawn upon the best ideas (claimed by Sun) from the already existing tools. An advantage of this is that JPA is embedded in Sun Java distribution packages and will probably be a de facto standard as it is available to java developers by default.

2.5 Web Services

In recent years the need for different programs to communicate has increased considerably as more applications are required to communicate in one way or another to fulfil the user’s needs; hence the need for a standardized frame for this communication across programming languages has arisen. A more or less obvious choice of base language for this was XML because of its broad acceptance as standard language for representing and transmitting structured data. Building on the broad acceptance of XML, Web services are applications that use standard transports, encodings, and protocols to exchange information (Microsoft, 2006).

XML provides the common syntax for representing data, SOAP[4] is used for data exchange and WSDL[5] is used to define the service provided by the web service (describes the methods provided by the web service application).

One of the most important attributes of this standard is composability, as shown in Figure 2 below.

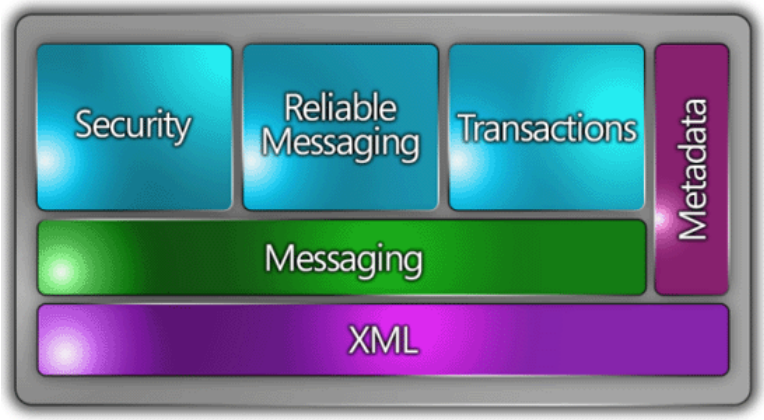

Figure 2: Schematic overview of the Web Service architecture (Microsoft, 2006)

Web services are built up of layers which can be included or excluded in actual implementations. Only the modules needed by the specific application should be implemented. This also gives the application designers the possibility of adding more features (like security for instance) later on if required.

[4] SOAP: Simple Object Access Protocol
[5] WSDL: Web service definition language

2.6 Java API for XML-Based RPC (JAX-RPC)

JAX-RPC is a java framework for developing applications that communicate with web services. A client generates the required classes according to the published (via WSDL) service from the server. A local stub[6] is generated which is the local interface for the application, encapsulating all communication complexity from the developers. The stub then calls the JAX-RPC runtime system which in turn performs the tasks of wrapping and serialising the regular method call to a SOAP message (called envelope in SOAP).

After Web Services' popularity and usage increased, a new version of JAX-RPC was introduced by Sun, it was initially called JAX-RPC 2.0, but because of some confusion with RPC (remote procedure call) and web services it was renamed to JAX-WS (Java API for XML Web Services) to eliminate this confusion. JAX-WS is the current version as of this thesis.

2.7 Java Message Service (JMS)

Messaging is an approach to communication between software components or applications. The main goal of messaging is to achieve a *loosely coupled* system. With messaging the sender and receiver connects to some messaging service which performs the creating, sending, receiving and reading. The sender and or receiver is not required to be available at the same time (in contrast to Remote Procedure Call (RPC) for instance), and this is called *loosely coupled*. In fact the sender and or receiver do not need to know anything about each other as long as the format and destination is defined.

This approach is also asynchronous and enables modules within a system to continue their work after sending some information to another module. In enterprise applications it is common to choose messaging over some tightly coupled approach due to the above reason and the fact that not all modules of the system is required to be up and running in order for the total system to function at some level. However, the asynchronous behaviour introduces several problematic issues such as state synchronization and verification of happened-before relations. Not all tasks are suitable for asynchronous processing, and an enterprise application is likely to implement both approaches in different areas of the system. If one module requires a task to be performed at some other module before continuing, this request should be processed synchronously.

Java Message Service is a service that provides the functionality described above and a suitable API for developers to use (encapsulation of unnecessary details involved in messaging services). In addition to the features described above JMS also provides a reliable communication, meaning JMS ensures that a message is delivered once and only once.

Due to time limitation the use of messaging between components (as defined by TPC) in our implementation was omitted, and regular method invocation was used instead.

[6] Stub: a local representation of some external entity, either method or an entire class.

3 System setup

This implementation is based on DBT4, PostgreSQL, JBoss and Hibernate. DBT4 (The Linux Foundation, 2006) is an open source implementation of the TPC application server specification. PostgreSQL is used as database for this system and as front-end application server we use JBoss with Hibernate integrated.

To be able to perform extensive timings on the system a second database was created to hold logging information. This database was kept completely separate from the application server to minimise influence.

3.1 TPC- App specification

In (Transaction Processing Performance Council, 2005), the TPC-App benchmark is characterised as follows:

TPC Benchmark™ App (TPC-App) is an Application Server and Web services benchmark. The workload is performed in a Managed environment[7] that simulates the activities of a business–to-business transactional application server operating around the clock. TPC-App showcases the performance capabilities of application server systems. The workload exercises commercially available application server products, messaging products, and databases associated with such environments.

The workload was designed specifically to stress the Application Server. As such, the work to be performed by the database was purposely minimised. Additionally, the application was designed such that it would cluster in a manner that is as nearly linear as possible. All application server systems are required to have identical hardware and software configurations. The workload is then distributed across all application server systems.

TPC-App does not benchmark the logic needed to process or display the presentation layer (for example, HTML) to the clients. The clients in TPC-App represent businesses that utilize Web services in order to satisfy their business needs. TPC-App does not represent the activity of any particular business segment, but rather any industry that must market and sell a product or service over the Internet via Web services (e.g., retail store, software distribution, airline reservation, etc.). TPC-App does not attempt to be a model of how to build an actual application.

The application portrayed by the benchmark is a retail distributor on the Internet with ordering and product browsing scenarios. The application accepts incoming Web Service Requests from other businesses (or a store front) to place orders, view catalogue items and make changes to the catalogue, update or add customer information, or request the status of an existing order. The majority of requests generate order purchase activity with a smaller portion of requesting item catalogue information. (Transaction Processing Performance Council, 2005)

[7] Managed environment: a software abstraction layer that sits between application code and the Operating System. It provides a logical runtime environment that insulates the application code from the native Operating System.

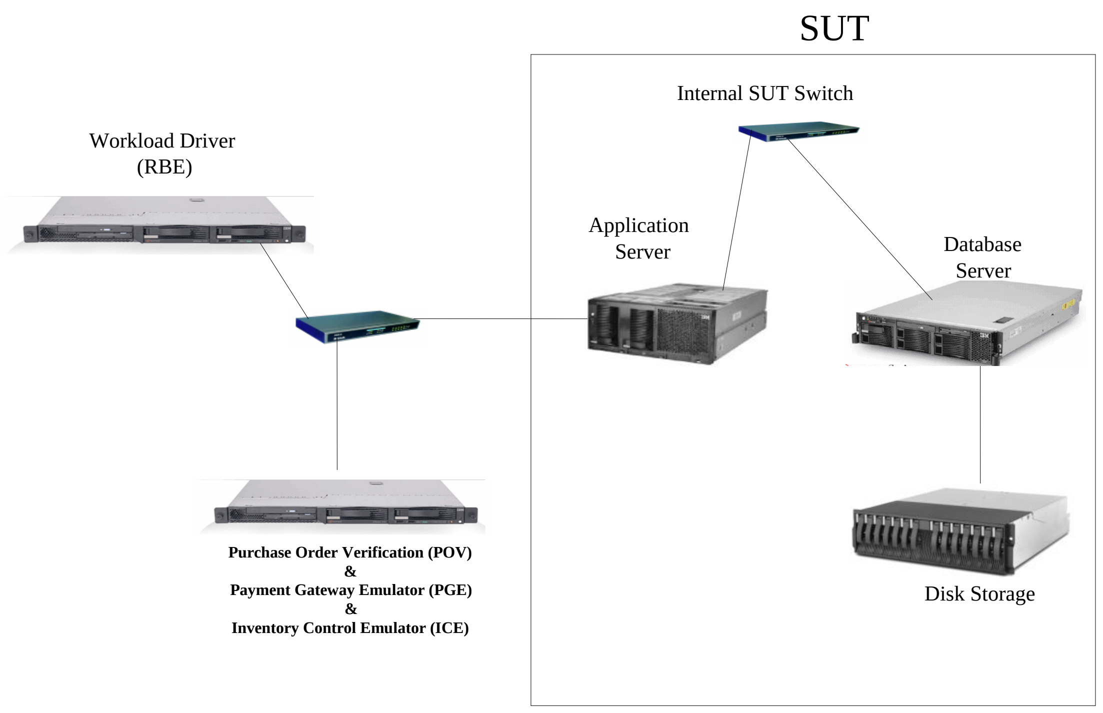
SUT
Internal SUT Switch
Workload Driver (RBE)
Application Server
Database Server
Purchase Order Verification (POV) & Payment Gateway Emulator (PGE) & Inventory Control Emulator (ICE)
Disk Storage

Figure 3 Total system schematic

Figure 3 shows a graphical representation of the entire System Under Test (SUT). In this implementation of the TPC-App specification some alterations to the above model were made. Firstly, the remote business emulator application was executed within the application server to minimise network latency influence (this way a minimal and stable latency between the remote business and application server is guaranteed). Secondly, the *Purchase Order Verification (POV), Payment Gateway Emulator (PGE),* and *Inventory Control Emulator (ICE)* were omitted in this implementation due to lack of time. These functionalities simply perform a static sleep of 500 ms on the application server. In addition to the setup displayed above, a configuration where both the application server and database server runs on the same physical machine (called mixed server as opposed to dedicated servers), is tested in this thesis.

3.1.1 Business methods

TPC-App simulates a user pattern with the following defined business methods:

1. **NewCustomer**: Creates a new customer in the database.

   Sends a request to Purchase Order Verification (POV) if payment_method passed by client is PO. Stores address information if not already present in database. Stores customer information in database and returns the new customerId.

2. **ChangePaymentMethod**: Changes payment method of existing customer.

   If payment method passed is PO then sends a request to POV and proceeds if approved. Updates payment method and returns current payment method

3. **CreateOrder**: Creates an order for an existing customer.

Obtaining payment method from customer information stored, if payment method is CC Payment Gateway Emulator (PGE) is invoked to validate credit card. Insert shipping address if not present in database. Calculate shipping cost of items. Obtain discount information from customer. Iterating over all items defined in client request and set order status to PENDING or BACK_ORDERED according to item stock. Calculate tax and totals for order and store order in database. Queue order in shipping queue and returns order details to client.

4. **OrderStatus**: Returns information about the last *n* orders placed by the customer.

   Authenticates user and proceeds if successful. Retrieves customer information. Retrieves last *n* (order count is passed by client) orders for current customer and returns order information for the orders.

5. **NewProducts**: Returns information about newly registered products.

   Retrieves items with published date newer than *cutOff* (between 1 to 10 minutes, passed by client) subtracted from current date and limiting maximum result to *item_limit* (passed by client).

6. **ProductDetail**: Returns detailed information about a set of items specified by a set of item ids by client.

   Gather detailed information for each item specified by client and returning information to client.

7. **ChangeItem**: Updates published date for a specified set of items.

   Retrieve each item specified by client and set *pub_date* to current date and returns all item ids with their new *pub_date*.

The emulated businesses distribute their business method requests as Figure 4 below shows. An emulated business runs a continuous run of business method requests (business session), selected according to the cumulative distribution function between 1 and 120, and then immediately starts a new business session with a new customer ID (this to ensure that data accessed in the database are evenly distributed over the entire dataset). The emulated business does not implement any keying time[8], this is because this system simulates a business-to-business system, and the client business will in most cases have several users of its system, resulting in a continuous set of requests to the application server.

[8] Keying time: an estimated sleep time to simulate human interaction with a computer (e.g. keying in information in a form or time to scroll over a set of products).

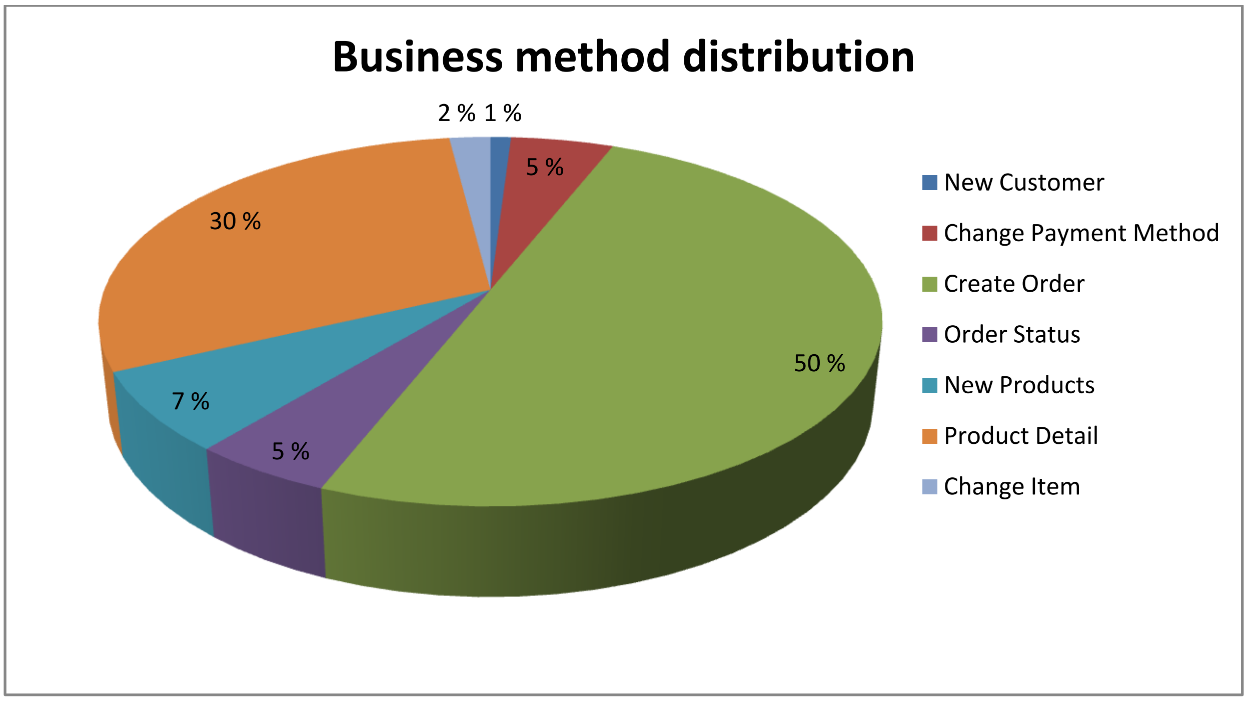

Figure 4: Business method distribution

3.1.2 Business methods resource profiles

After analysing the logics of the different business methods a rough resource profile estimate is listed below:

1. **NewCustomer:** is mainly database intensive and little processing is done by the application server, but is dependent on *Purchase Order Validation* which is an external web service not included in the SUT resulting in some idle time.
2. **ChangePaymentMethod:** little logic is performed here, but also dependent on *Purchase Order Validation* and some idle time occurs. Cheap on both processing and data access.
3. **CreateOrder:** most complex method among the business methods, some data is accessed (customer and items included in the order request), but is mainly processor intensive. Dependant on the external *Payment Gateway Emulator* which also results in some idling.
4. **OrderStatus:** the most data intensive method, includes all the largest tables in the database (orders, order_line, items and stock), heavily loading the database server. Little logic is involved, application server not stressed.
5. **NewProducts:** practically no processing required by the application server, all processing performed by database server (simply listing all items newer than the specified date). The items table is the largest table in the database resulting in a heavy load on the database server.
6. **ProductDetails:** similar resource profile as *NewProducts*, but on average more records are returned from the database resulting in a slightly more costly method overall.
7. **ChangeItem:** little load on the application server, simply updates a set of records in the item table, most processing performed by database server, and most of this processing is data IO.

3.2 Implementation issues

The initial purpose of this thesis was to study the practical impact of Sun's radical change in entity management between EJB 2.1 and EJB 3.0. Due to the need of a standardized benchmark system the TPC-App implementation DBT4 was chosen, but after a short time it became evident that DBT4 was not finalized in any way and just getting this benchmark up and running became a large part of this thesis; Hence, the planned comparison between EJB 2.1 and EJB 3.0 were considered too extensive for this project. Instead, we decided to profile, and if possible improve, the performance of an EJB-3.0- based stack running a TPC-App compliant benchmark.

DBT4 is not in any stable release version and the amount of additional coding in order to get the DBT4 implementation to work was close to a complete rewrite. Several aspects of the DBT4 implementation did not comply with the TPC specification. Two of the major deviations were:

- All communication between the application server and the remote emulated businesses was performed using java rmi instead of web services. This makes DBT4 hard to compare to any other implementations as these two technologies differs drastically in how the endpoints communicate, this communication layer was completely rewritten.
- All RBEs (Remote Business Emulator) had static business session length (the number of business methods to invoke). The RBE should perform a random number of business methods according to the cumulative distribution function (See appendix C in TPC specification), and after one session finishes another is started immediately after. This is to ensure that the data accessed by the RBEs are evenly distributed within the database (a new customer id is chosen in each session).

The deviation from transport technology, web services, would prove to be the most challenging part to implement. Web services are not a mature java technology and there exist some different implementations of this specification. After some research and painful hours of debugging it was clear that JBoss does not support java 1.6 (when hosting web services) nor does it support JAX-WS (JAX-RPC's successor, see section 2.4). AXIS2 was eventually tested, and this implementation works with JBoss. Apache Axis2 is the core engine for Web services (The Apache Software Foundation, 2007) , and its web service definition language (WSDL) generation tools (WSDL2Java and Java2WSDL) did work with JBoss.

The other deviations in DBT4 were overcome with less effort, but the time spent reading and understanding the logic of others code in a project of this magnitude was considerable. And none of the business logic was implemented so the entire session bean containing the business logic had to be implemented in addition to all entities.

Another problem arose when implementing the monitoring code for the JDBC driver; this is done to get an estimate on how much of the total time is used waiting for the database. In order to avoid altering the entire Hibernate API and PostgreSQLs JDBC API some sort of common (between the JDBC driver and the business methods) identifier is needed to be able to determine what method performed this JDBC call. The only common knowledge between the JDBC method and the session bean is their thread; hence threadId was used to identify what business method performed each JDBC call. This worked fine during debugging (single RBE performing single business method call), but after reviewing the results from these timings in a real test it was evident that this approach cannot fully be trusted. The reason for this is that JBoss is issuing several threads while working, and no documentation on how JBoss and Hibernate handles threading during execution was found. This does not mean that the timings are all wrong, but there is no guarantee that all the time spent in the

JDBC driver is included; some of the JDBC calls may have been performed within another thread. Another alternative to this approach would be to use reflection to get a reference to the caller of a method invocation; however, this is not supported by java reflection. This lack of functionality was submitted as a question to the java community, but the only response was that the need for this kind of functionality is evidence that the implementation of your application is wrong, a somewhat strange argument. This functionality is supported in .NET C#, the main competing programming language to java, and is used heavily in .NETs event handling.

After running tests for some period of time a strange result arose, *NewProducts* was extremely inefficient and shot through the roof of every graph. At first this was assumed to be because it accessed the largest table in the database, items (100k records), but this did not explain why it was more costly than *orderStatus* which also queries the item table along with other large tables (*NewProducts* only uses item table). After some further analysis and painful despair over this anomaly a glitch was found, Java counts months from 0 not 1 which sql does; this resulted in *NewProducts* returning *Item_limit* of new products each time instead of only the items newer than 1-10 minutes (randomly selected). What charm would a software project have without bugs!?

3.2.1 Functionality not implemented

Sun's JPA standardized API should have made it possible to simply plug in and plug out different object relational mapping tools, and in the initial project description a comparison between the object relational mapping tools Hibernate and TopLink was included. However, this proved not to be possible. JBoss ships with Hibernate as object relation mapping framework. EJB3 has recently been released in its final state, and JBoss 4.0.5 does not fully support this specification yet (EJB3 will be fully supported in version 5.0), this resulted in various errors when Hibernate was replaced with TopLink in JBoss. After some discussion on the JBoss forum, TopLink was excluded from this thesis as version 5 of JBoss was not released and no one seemed to have successfully configured JBoss with TopLink[9] in version 4.0.5.

TPC-App specifies that all method invocations on the application server should be performed using reliable and durable messages (in java this would be JMS) to ensure system consistency in case of system failure. DBT4 did not implement this (as none of the business logic was implemented), and this was excluded from the project due to time limitation. Instead, regular method calls are used within the business logic on the application server.

The external vendors (such as Inventory Control Emulator (ICE) and Payment Gateway Emulator (PGE)) modules were not implemented by DBT4 and these functionalities were beside the main focus of this thesis, thus merely set to a static sleep of 500 ms.

3.3 Test configuration

3.3.1 Logging

In order to provide useful performance measurements timing is performed on several levels throughout the system, both on client side and server side. Server side timings are performed in the web service class as well as within Hibernate. All these timings are logged to a database running at an external server to minimise influence by logging, to further minimise this effect all logging is performed batch style. This minimises logging influence to merely log object creation (logging will

[9] JBoss persistence provider does not implement EJB3 specification correctly, *getNewTempClassLoader()* not implemented!

always have some impact on the actual system being logged) and avoiding the costly persisting phase to interfere with the actual timing.

To describe how logging is performed on the server an explanation of how a client request is handled at the server, and how the server side application communicates with JBoss and Hibernate is needed. An enterprise application (server side application) is most often implemented using a session bean (session beans are extensions of the client application that manage processes or tasks (Burke & Monson-Haefel, 2006)) and a set of entity beans (representing data stored in the database). Figure 5 JBoss components communicationshows the communication path from clients to the application server and how JBoss and Hibernate is built up from a session bean's perspective.

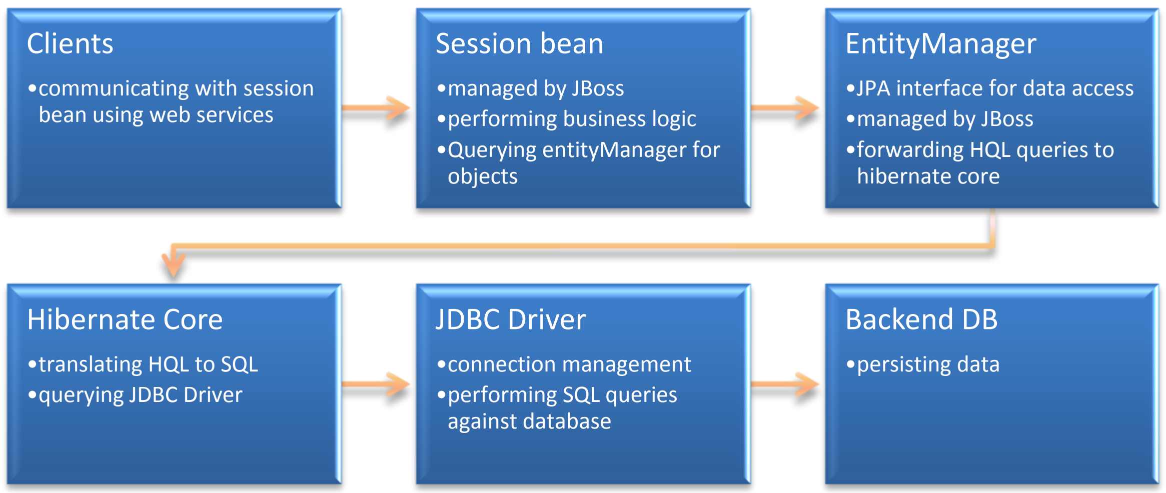
Clients
•communicating with session bean using web services
Session bean
•managed by JBoss
•performing business logic
•Querying entityManager for objects
EntityManager
•JPA interface for data access
•managed by JBoss
•forwarding HQL queries to hibernate core
Hibernate Core
•translating HQL to SQL
•querying JDBC Driver
JDBC Driver
•connection management
•performing SQL queries against database
Backend DB
•persisting data

Figure 5 JBoss components communication

In the Hibernate stack timings are performed on six different JBoss' *EntityManager* calls: CREATEQUERY, CREATENATIVEQUERY, PERSIST, FIND, GETRESULT and GETSINGLERESULT. In addition to these timings, the JDBC driver is monitored to find the total time Hibernate is waiting for the database.

3.3.2 Hardware

Two scenarios were analysed in this thesis, *mixed mode* and *dual server mode*. Mixed mode consists of the system described in section 3.1 with both the application server software and database running on the same physical server. This scenario represents smaller businesses where cost is the primary limitation, and implementing a system with two or more servers would be an overkill. The second scenario is dual server mode; here the application server software is running on one server and the database on another server, communicating over an Ethernet link (this scenario can easily be extended for further increase in performance, 2 application servers sharing the workload between them for instance).

Below the hardware configuration for the computers involved is specified

Mixed Server (JBoss and PostgreSQL running on same server)	
Number of CPUs	2
CPU clock speed	2,2 GHz
RAM	2 GB
OS	Windows Server 2003
Disk	SATA Disk (size=300GB, seek time=9ms, buffer=16MB, transfer speed=300MB/s)

Dual Server (JBoss and PostgreSQL running on separate servers)	
Number of CPUs	2
CPU clock speed	2,4 GHz
RAM	2 GB
OS	Red Hat Enterprise Linux Advanced Server 3
Disk	Dell PowerEdge Expandable RAID Controller 3/Di, 5 SCSI disks in raid 5 (4 spindles, 1 hot spare)

4 Test Results

The testing is performed by increasing the amount of emulated businesses and see how the system reacts to this increase. For the hardware available for this thesis the test sequence was set to 2, 5, 10 and 20 parallel emulated businesses (more than 20 was pointless because the server had reached its saturation point, see results later in this chapter).

Two metrics are used to measure system performance: Service Interactions Per Seconds (SIPS) and response time of each business method.

There are three main optimisation areas addressed in this thesis: handling of binary data, see section 3.1 (Items), memory tuning and query optimisation (optimising of database schema). An initial test run with no alterations done, both JBoss and PostgreSQL running with default setup, is performed and an incremental tuning is performed to see what performance effect each tuning results in.

4.1 Mixed server

Both JBoss and PostgreSQL are running on the mixed server described in section 3.3.2 in this initial run. First analysis involved measuring how response times developed when the number of parallel emulated businesses increased.

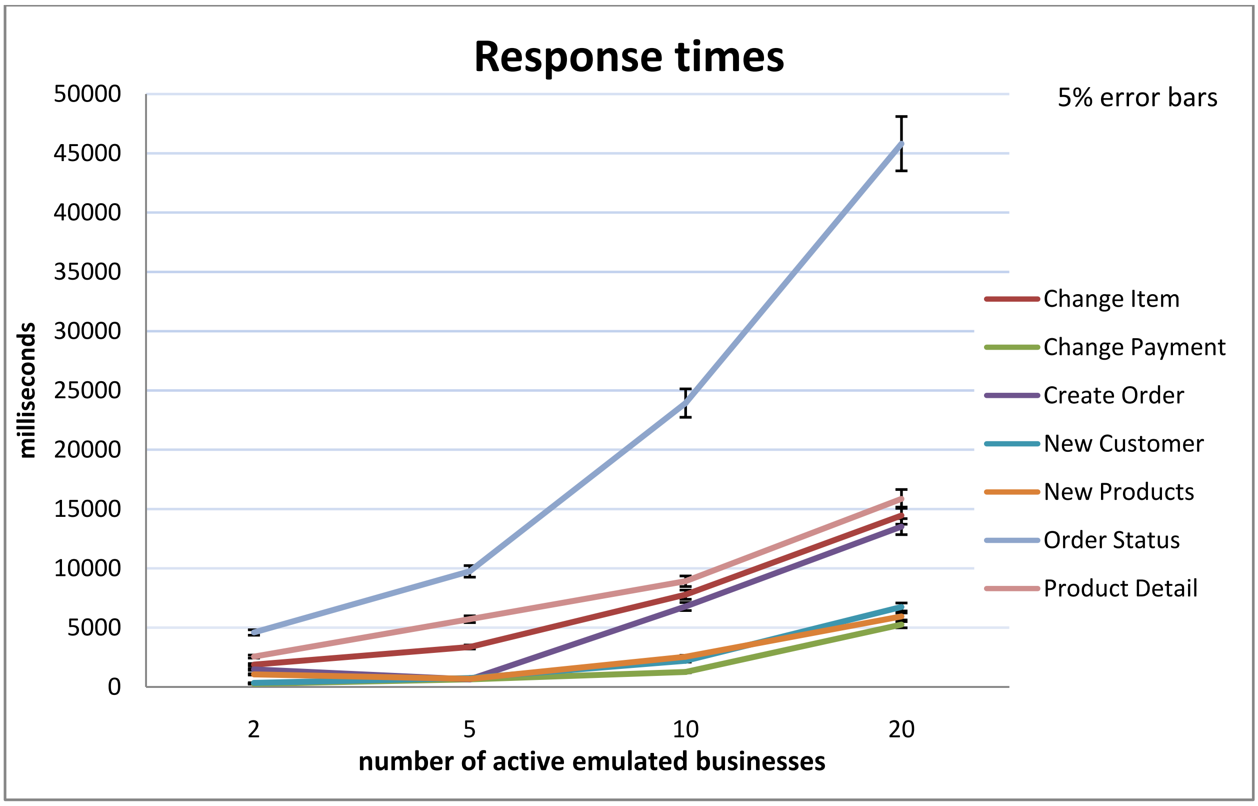

Figure 6 Response times

As expected the response times increases by the number of parallel businesses querying, but the unusually high response times were unexpected. Another diagram that illustrates this poor performance is the Service Interactions per Second (SIPS) by emulated businesses showed below.

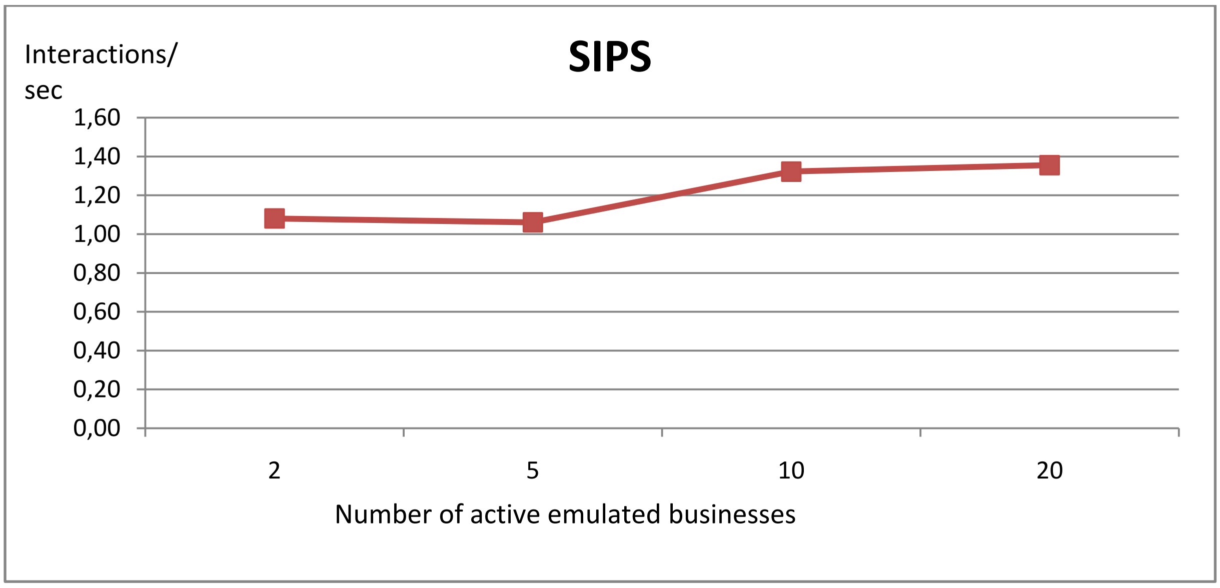

Figure 7 Service interactions per second

It is clear that a system that only manages to serve a little over 1 service interaction per second is inadequate for most users. To find out what resources get depleted and causing this poor performance an analysis of some key system performance counters is required. There are three likely candidates: processor, memory and/or disk. An analysis of the processor quickly reveals that this is not the bottleneck; the processor is not running at 100% at any time during all four test runs. System memory is also sufficient (even if memory was of somewhat shortage, this would not explain such a low performance), disk IO however is over stressed by far. Microsoft states that an average disk queue length of 2 or more means the disk is over stressed and is a bottleneck in the system. In this test run the average disk queue length was at all times above 2 and peaked close to 6 as Figure 8 shows.

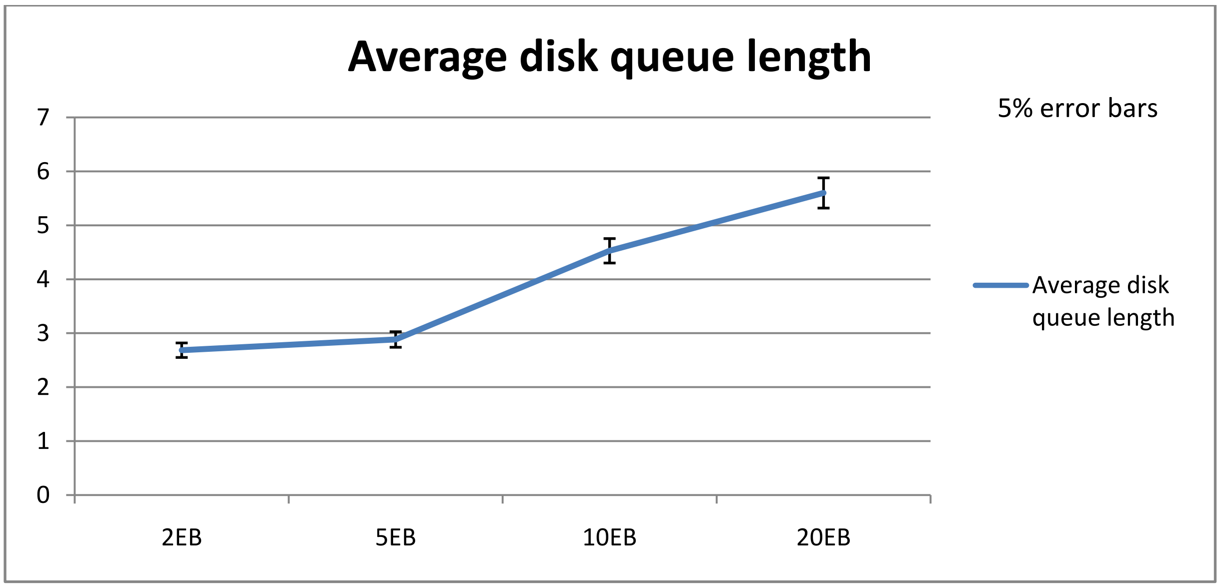

Figure 8 Average disk queue length

Further analysis revealed that a majority of these read/write requests came from PostgreSQL. Therefore a decision to reveal the database of the binary data (image data of items) was made, replacing it with a reference to an external image file.

4.1.1 Binary data alteration

A new set of test runs were performed, now with item records referencing external files instead of storing the actual binary image data. This results in all operations performed on item objects that does not need the image data does not load this data from disk, in contrast to what happened when the binary data was stored in each item record, hence loaded by Hibernate every time an item object was queried (even if one only required price information about the item). When the image data is required the enterprise application loads the data from disk using the reference stored in the database.

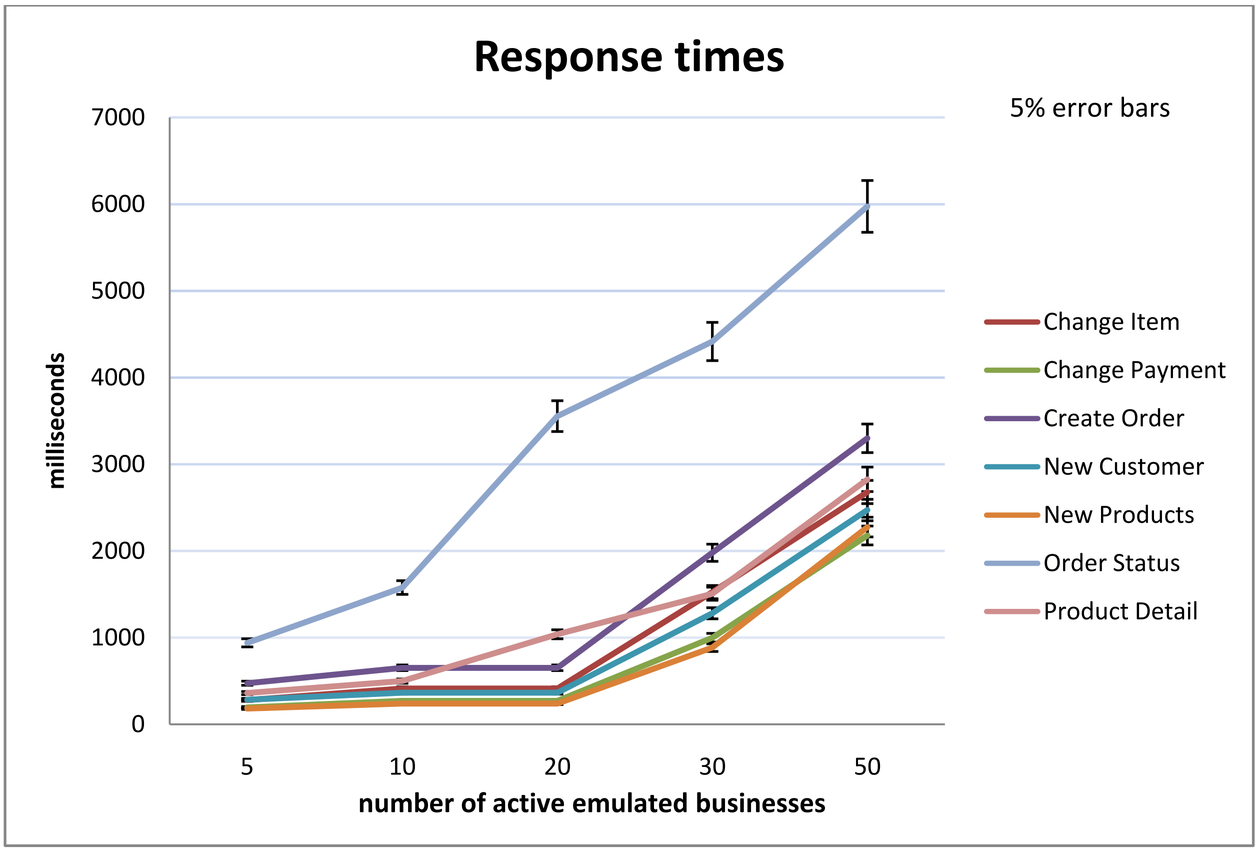

Figure 9 Response times without binary data

Figure 9 shows that this configuration has reduced the response time to approximately 10% of the response time experienced with binary data stored in the database (see Figure 6). This also gives a completely different throughput as Figure 10 shows, also changed by a factor 10.

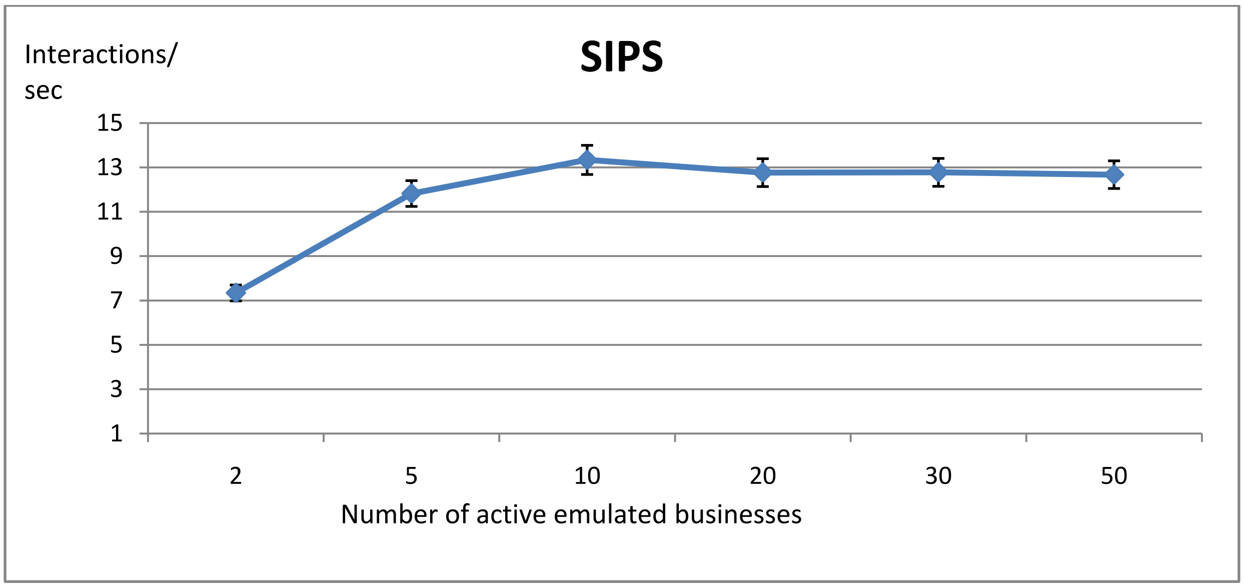

Figure 10 throughput without binary data in DB

4.1.2 Memory optimisation

PostgreSQL can be tuned via numerous parameters a configuration file. The first surprising fact after talking to several members of the PostgreSQL community was that there was no agreed level on the various settings on a given hardware. The answers agreed on one thing and that was that there exists no absolute answer to these values as the results are too complex to get a complete and precise set of recommendations. However, after some discussion it was clear what range one should try on most variables.

4.1.2.1 Internal memory and IO parameters optimisation

These are the tuning parameters of most interest when tuning PostgreSQL for performance in our usage scenario:

- ***shared_buffer****:* is the dedicated memory for active operations. This value should not be set too high as this will reduce the overall performance. The reason for this is that PostgreSQL relies heavily on the operating systems disk caching capability (see *effective_cache_size* below), and allowing PostgreSQL a too large amount of memory will reduce the operating systems disk caching. The exact value for this parameter is a very complex calculation of hardware specifications, dataset size, and usage pattern and most often a rule of thumb is used to estimate this value. Rule of thumb in this case is 10-15% of available RAM.
- ***effective_cache_size*****:** tells the query planner how much it can expect the operating system to cache. Raising this value will make PostgreSQL decide on index search over sequential search more often. The value for this parameter is difficult to calculate; it depends largely on what the server is dedicated to do; a database only server or a mixed server. If the server is a dedicated database server this value could be set to approximately 50% of physical RAM. If the server is a mixed server then one would need to estimate the other applications need of the operating systems disk caching and physical RAM and use the remaining available RAM as guideline.
- ***fsync:*** tells PostgreSQL whether it should perform an immediate synchronization to disk or allow some delay. The risk of allowing delay is loss of data. If the server unexpectedly crashes or stops for some reason the alterations to data that has not yet been synchronised to disk will be lost. On a production server synchronization delay could be accepted in most

cases because of very robust hardware, and in many cases a small data loss is not critical. Many professional server systems now has an up time guarantee of 99.99%, and also contains advanced failover features. In addition to minimising the probability of a hard crash, many applications can be developed in such a way that it tolerates data loss, i.e. if no response is received from the server (within a predefined timeout length), the client application can conclude that the request failed and inform its user about this unexpected behaviour, then the user can decide what to do with this.

To get an idea of what parameters give what results the tests are performed with incremental tuning of parameters. First stage is to set the *shared_buffer* and *effective_cache_size* to assumed proper values and see if this impacts the overall performance. *Shared_buffer* was set to 256MB and *effective_cache_size* was set to 512MB in the initial run, this gave no impact on the performance of the system. After looking at memory usage statistics during the test run, a second configuration was made, setting *effective_cache_size* to 1GB. Again, this setup resulted in no significant changes in performance.

The next parameter to tune is *fsync*. This was turned off (allowing delayed synchronization to disk) and the same test (*shared_buffer=* 256MB, *effective_cache_size*=512MB) was executed over again with no significant difference in performance. This could be supported by the low disk queue length achieved after relieving the database of the binary data; hence disk is not the bottleneck.

4.1.3 Query Optimisation

Since no significant increase in performance was achieved by tuning PostgreSQL's memory and IO parameters the next step is to look at the schema and what queries the enterprise applications generates. Since this test is performed in compliance to the TPC specification no changes to the schema layout is allowed, but in general there are many pitfalls to avoid when specifying a database schema. However, TPC does not enforce any limits on internal structure of relations in the schema, hence adding indices would not compromise the compliance between this database configuration and the TPC-App specification. Analyses of all SQL queries produced by the business methods were performed and several potential improvements discovered.

NewProducts creates only one single sql query. After analysing PostgreSQLs query plan for this single query it is evident why this query performs poorly. PostgreSQL builds a sequential scan for this query; this is very inefficient since this table contains 100 000 item records (sequential scans are more efficient than index scans on small tables). The reason PostgreSQL uses sequential scan is that the conditions introduced by the business method involves the *subject* field and *pub_date* field, there exists no index on these columns so no index scan is possible. An obvious solution to this is to create indices on these columns. In this scenario it is no doubt that adding indices will increase performance, but one should keep in mind that although adding indices improves fetching of data, it decreases updates and insertion of data.

OrderStatus on the other hand, performs more than one query; first it finds the customer via customerId (customerId is the primary key of customer, primary keys are always indexed), then it performs a query on the order table matching against the customerId column in orders. When analyzing this query PostgreSQL also performs a sequential scan, so the same solution should be applied here; i.e. adding an index on customerId column in orders. In addition to these two queries *orderStatus* also performs a query on the order_line table for each order it found in the query against the order table. The condition in this last query is matching orderId in order_line against the orderId of each order. OrderId in order_line also lacks an index, this column is indexed for the next test run, as well as the foreign key to the item table (ol_i_id).

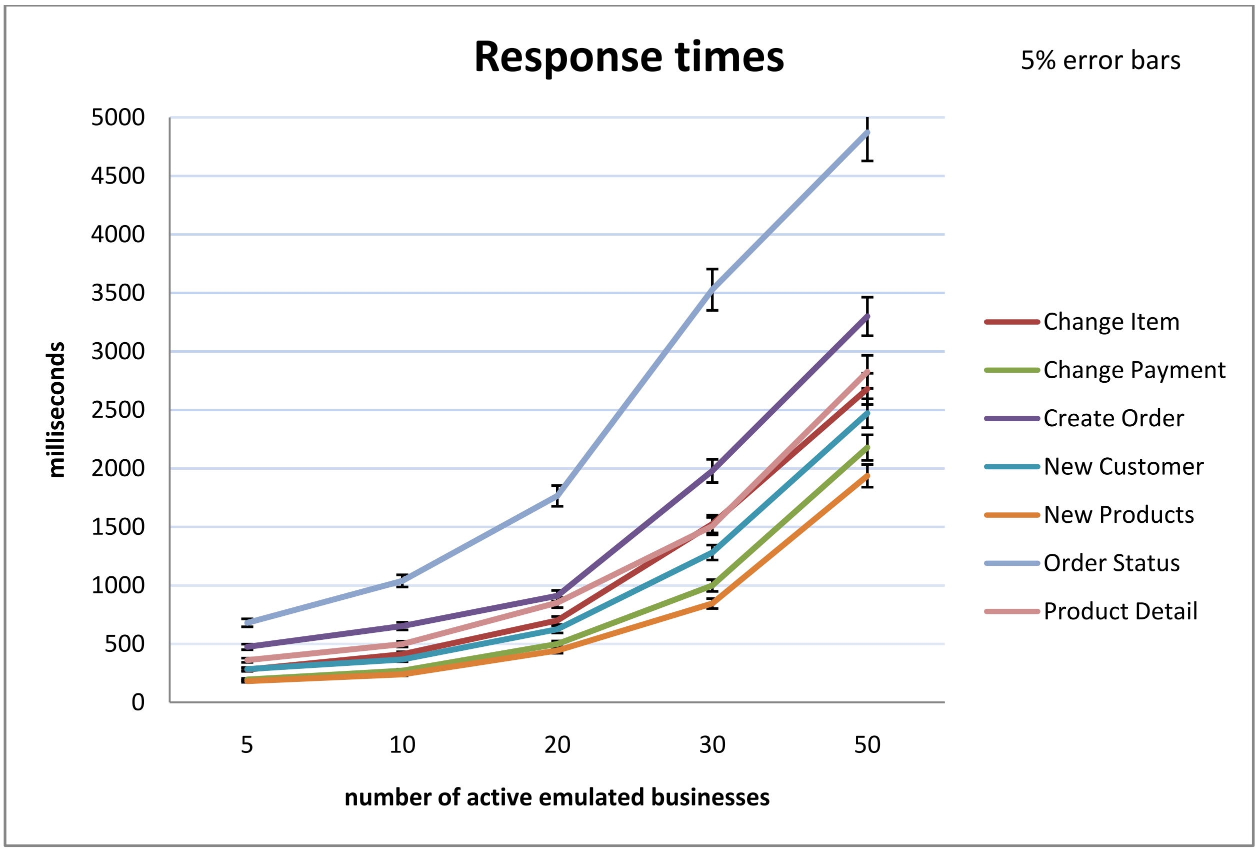

Figure 11 Response times after indices database is indexed

As Figure 11 shows, *OrderStatus* is now reduced by approximately 20% by the indices in the database. The reason *OrderStatus* still is so costly is caused by what tables this business method queries: when the queries from this method is executed *orders, order_line, item, author* and *stock* are involved and these tables are the largest tables in the database, hence the most costly lookups.

NewProducts also uses three of the largest tables (*item, stock* and *author*), but does not need to go via *order* and *order_lines* to get to *item* and further, as *OrderStatus* does. This explains the difference in response time between the two. However, only approximately 10% improvement in response time was achieved on *NewProducts* by applying additional indices to the item table.

These performance improvements also impacts total throughput of the system as Figure 12 Throughput comparison shows

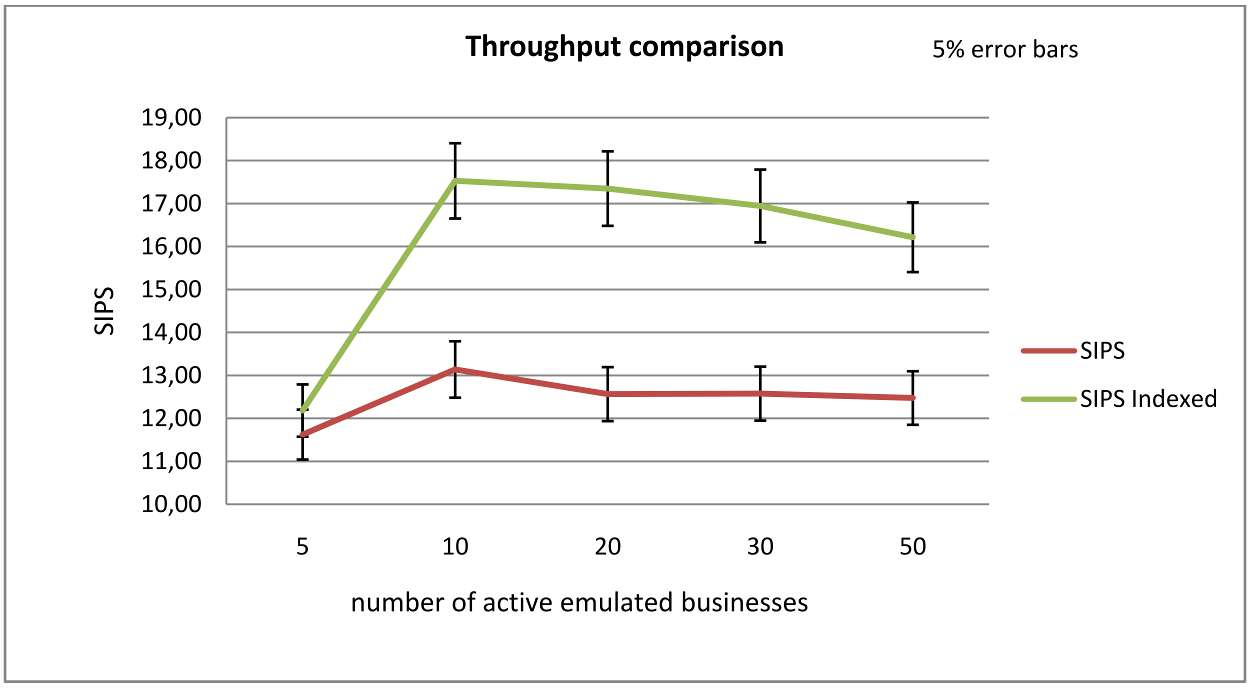

Figure 12 Throughput comparison

The total throughput of the system has increased by approximately 35% by the response time reduction achieved on *NewProducts* and *OrderStatus*.

In this configuration all optimisation and tuning[10] are applied so it is now interesting to see what resources are limiting the system. Disk queue length is now below the threshold specified by Microsoft and there are still some free memory left, this leaves us with the processor, and as Figure 13 shows the processor is running at maximum for 10 or more emulated businesses.

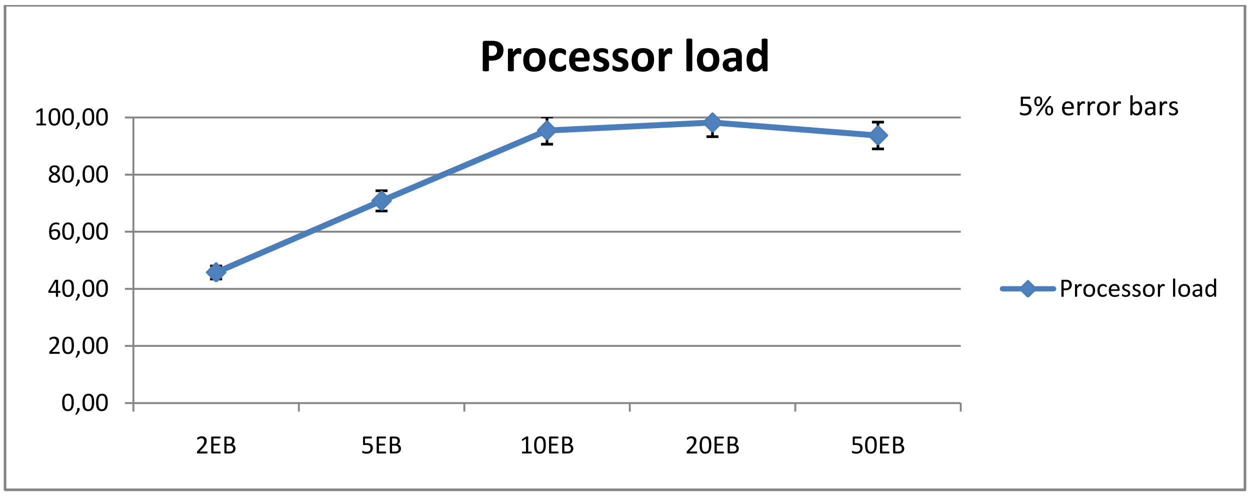

Figure 13 Processor load optimised configuration

[10] some further optimisation could be done, see further work chapter, but was not possible within the time frame of this thesis

4.2 Dual server mode

As described in section 3.3.2 this test run was performed with two servers, one responsible for the database processing and one responsible for the front end applications. The tests performed on this configuration used binary data separated from the relational database (see section 4.1.1), indices applied to the database (see section 4.1.3) and PostgreSQL with optimised memory settings (see section 4.1.2).

Due to the available hardware resources a slightly different machine configuration was employed when running in dual server mode (see section 3.3.2). A direct comparison between the dual server configuration and the original mixed server configuration would not be a fair comparison, so the extended tests were executed over again on one of the machines involved in the dual server configuration to give a precise comparison.

4.2.1 Enterprise Linux vs. Windows Server 2003

The machines used in dual server mode are assumed to be slightly more powerful than the single machine used for the single server setup; however, after running the extended tests on one of these machines the results were surprising. The total performance of the system was considerably lower than the machine running with less powerful hardware as Figure 14 shows.

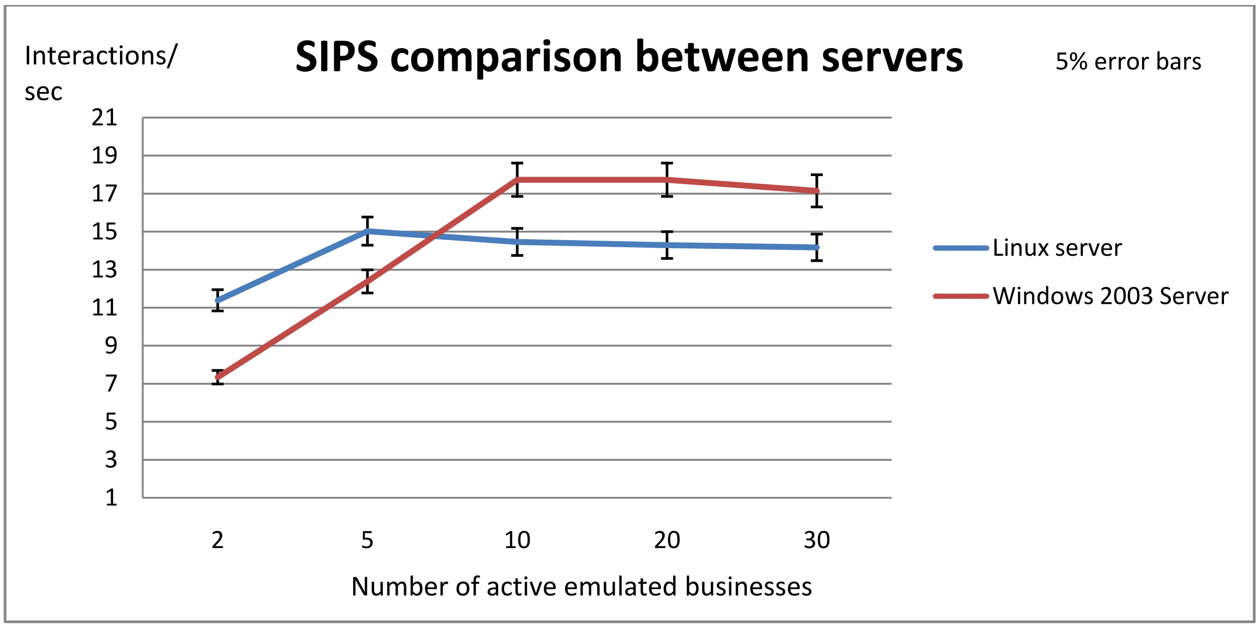

Figure 14 SIPS comparison between servers

The Linux server is saturated at 5 emulated businesses where as the windows server reaches saturation at 10 emulated businesses, and also performs 26% better on average when saturated. With a considerably faster disk configuration and slightly faster processors the likely cause will be the operating systems; further analysis on this is complex and was not possible within the time frame of this thesis.

4.2.2 Throughput

Now with a complete test run executed in mixed mode on one of the two servers used in the dual server mode a direct performance comparison can be done to visualise the impact of running both the application server and database server on one machine in contrast to separating these.

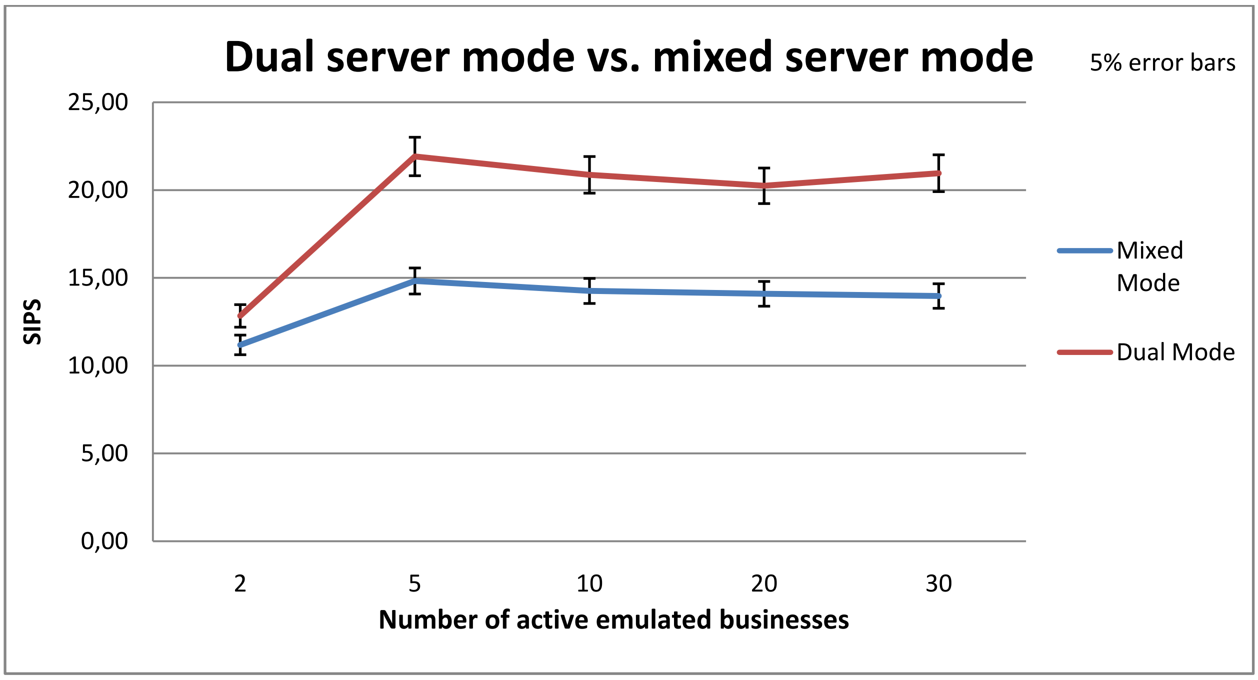

Figure 15 Dual server mode vs. mixed server mode

Figure 15 shows a comparison between a server running both the application server and database server (mixed mode) and two servers (one responsible for the application server and the other for the database) communicating; it is clear that performance increases significantly when running in dual server mode. It appears that in both cases the system is saturated at approximately 5 EBs, but in dual mode the system performs 32% better on average when saturated (EBs >= 5), and a peak performance of 22 SIPS is achieved in dual mode.

This difference in performance is achieved by the application server and database server not competing over CPU and IO resources (although the application server requires very little IO), and because of the very low network latency between the servers (if this latency was significantly higher than the local latency between the application server process and database process then this could have cancelled out the benefit) a total performance increase is seen.

4.2.3 CPU and memory analysis

Figure 16 and Figure 17 show how the application server and database server consume CPU and memory resources. The application server is both more memory and CPU intensive; the application server performs all the business logic; hence the higher CPU usage. Figure 16 and Figure 17 also show that the total need for CPU would exceed 100% if they were running on the same machine as is the case in mixed server mode; this reduces total performance of the mixed server compared to dual servers.

Another interesting result is that the processor is maximised on the application server from between 5 and 10 active emulated businesses; this suggests that the processor is the bottleneck of the system when running on dual servers. The same result was found when running in mixed mode.

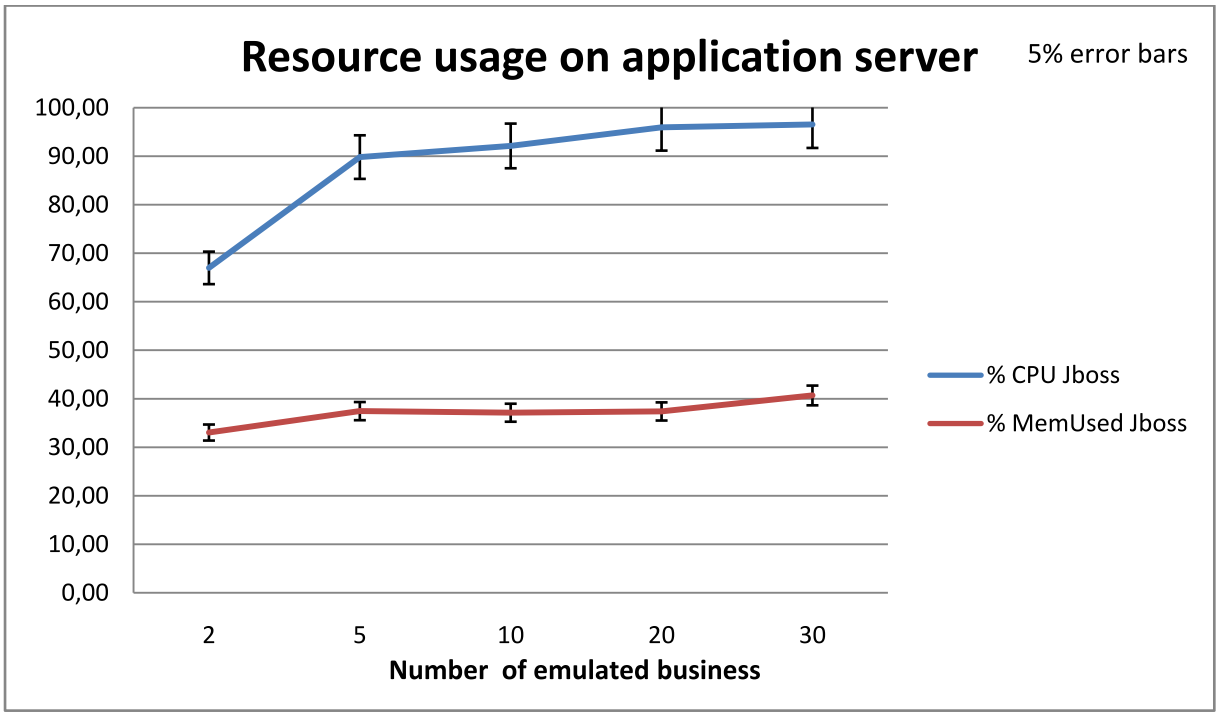

Figure 16 Application server CPU & memory usage

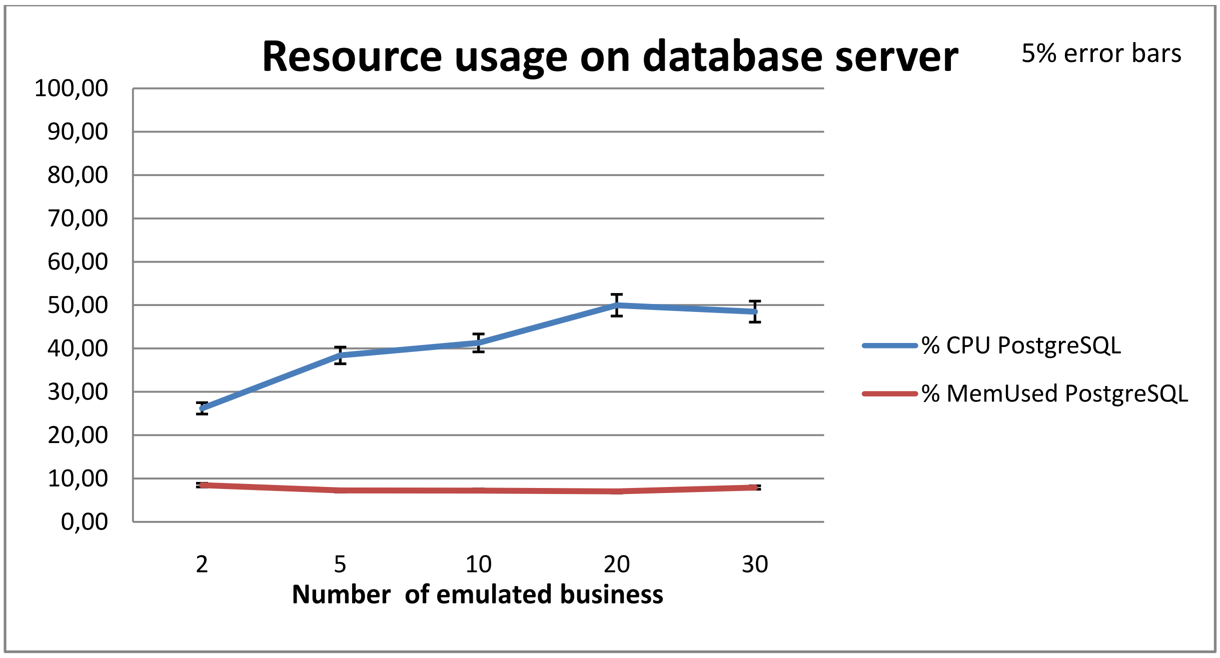

Figure 17 Database server CPU & memory usage

The reason PostgreSQL uses relatively little memory is because it relies heavily on the operating system's disk caching[11] feature instead of allocating memory for this itself. The operating system uses available memory for disk caching, and this feature is essential for PostgreSQL's performance. Figure 18 shows how much memory is reserved for PostgreSQL on the database server and how much memory is reserved for both PostgreSQL and JBoss on the mixed server, the operating system has 95% available memory for disk caching on the dedicated database server, but only 70% available

[11] Disk caching: is used to increase performance of physical disks. The operating system stores the last accessed data in memory, this allows programs that access same data over again to read it from memory instead of rereading it from disk

on the mixed server. This results in PostgreSQL reading more often from disk than memory when running in mixed server mode. To illustrate this Figure 19 shows how disk bandwidth usage develops during a test run (20 active emulated businesses was chosen in this graph); during the warm up phase the disk bandwidth is maximised, because none of the data required by the business methods have been loaded yet, but as time goes disk bandwidth usage decreases. This is because the operating system's disk cache contains more and more of the data queried, hence no need for disk IO. When data queried is stored in disk cache it is called a page hit, and when it is not stored in disk cache and a disk read is required it is called a page fault.

Another interesting information provided by Figure 19 is that the mixed server utilises about 15% more of the disk bandwidth than the dual server when the system has stabilized; this correlates to Figure 18 which shows that the operating system has less free memory for disk caching on the mixed server than on the dedicated database server; hence more frequent page faults (and more disk bandwidth usage) on the mixed server.

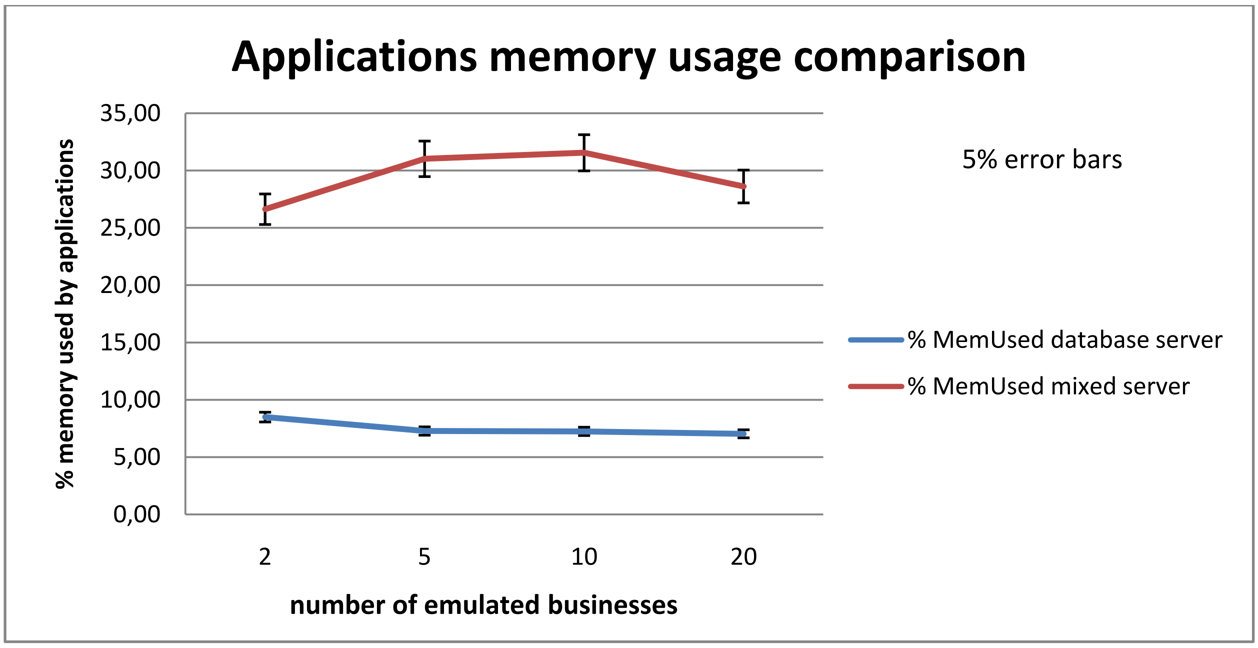

Figure 18 Applications memory usage comparison

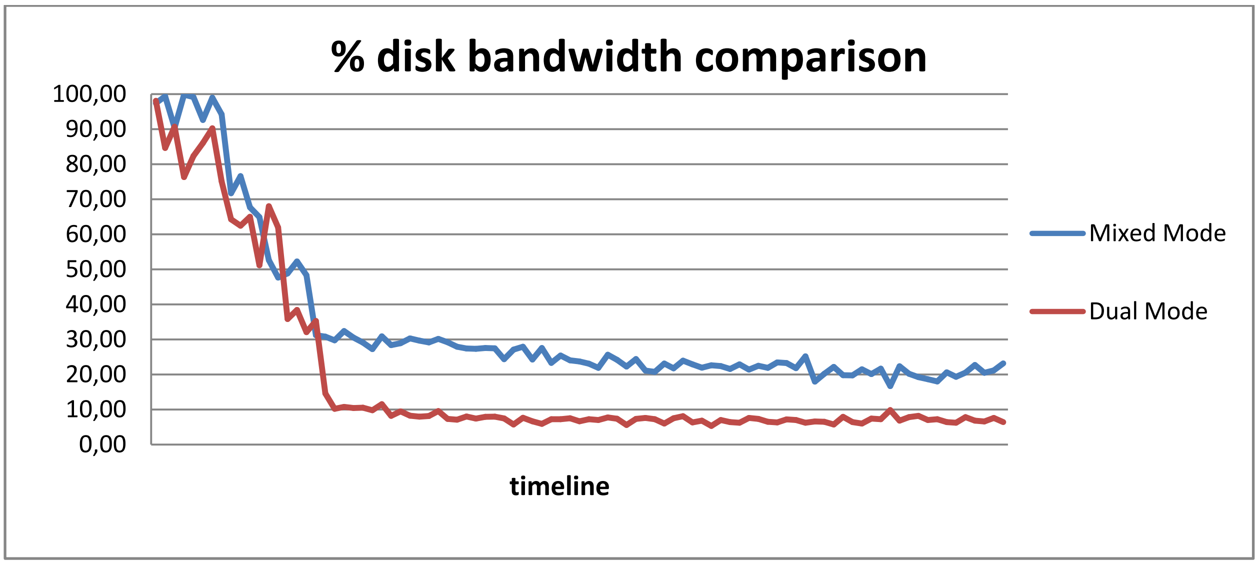

Figure 19 Disk bandwidth usage, mixed mode vs. dual mode

5 Conclusion

The focus of this thesis was to inspect a java enterprise application stack, primarily focusing on the persistence and object relational mapping performance. Hibernate was used as object relational mapping, but unfortunately it was not possible to get Oracle's TopLink to work with JBoss and the comparison between TopLink and Hibernate was excluded from this project.

After initial execution of the system specified by TPC without any tweaking or tuning an extremely poor total performance was seen. After analysing this the binary data was separated from the database and stored as regular files in the file system instead. This resulted in the most significant increase in performance of all the optimisations conducted. The reason for this increase was that each time an item was queried (no matter what information about that item was required) a record of size ~100KB was loaded as opposed to ~1KB without the binary data; this was supported by the disk queue length also shortened by a factor 100. JBoss provides some loading optimisation options for scenarios where one can specify different loading schemes that can be used by different queries and thereby only loading the interesting fields of a database record. However, this is only supported for EJB version 2.0 and 2.1 (in these versions JBoss is responsible for the object relational mapping) since Hibernate is responsible for the object relational mapping in JBoss when running EJB version 3.0. The Hibernate core supports query optimisation such as different fetch strategies, but since Hibernate is accessed through the generic JPA defined *EntityManager* interface much of these optimisations are unavailable, resulting in the entire record getting loaded each time. JPA need to address this problem in further releases of the EJB framework.

Before the testing started, I had much faith that memory optimisation of PostgreSQL would result in significant performance increase because the default memory settings in PostgreSQL is very conservative, but after all the buttons and screws were pushed and turned, little if any performance difference could be documented. This was surprising and some time was spent talking in the community about these settings and their impact on the system. The conclusion from this was that there are no definitive values for these settings (a full analysis of memory settings alterations and their impact is said to be too complex) and that the most critical memory aspect is how much memory is available for disk caching.

Indices in databases have a well known improvement on performance (typical index algorithm used is B tree, which results in logarithmic amortised time) and in a system where the query pattern is defined beforehand (human beings do not directly query the database) this is an obvious and cheap optimisation. Queries generated by the system should be analysed in detail by inspecting the query plan of each query, this gives a precise estimate as to where time is spent during execution and indices can be applied accordingly.

To summarise the two most effective optimisations carried out was separating binary data from the database and applying indices according to queries performed by the enterprise applications. When running this optimised version both in mixed mode and on dual servers the processor on the application server was the limiting resource, accordingly it is reasonable to conclude that adding more processor power and/or adding more application servers that divide the work between them would increase system performance until the database server is saturated and becomes the bottleneck. The hardware specification for the database server is not very powerful and by increasing memory and processor power on this machine (could also replicate this data and add a replication manager) in addition to adding more front end application servers would increase performance considerably.

6 Further work

As described in this thesis there are a few aspects of this implementation that do not fully comply with the TPC-App specification (due to limited time), these parts of the system should be implemented to get a performance result that can be fully compared with other implementations of the TPC-App specification. The parts not implemented are:

- **Messaging**: The use of messaging for method calls performed by the business methods is required to be done by durable messages (messaging), to ensure that the running system will withstand an unexpected abortion of the application server and be able to recover when started up. There are some overhead involved in messaging compared to direct method invocation and this will impact total system performance and should be implemented in future versions of this system.
- **External Vendors**: are the functionality shown in Figure 3 as POV (Purchase Order Verification), PGE (Payment Gateway Emulator) and ICE (Inventory Control Emulator). These components simulate how a business performs an external call to a functionality required to provide the services to its customer (e.g. verification of credit card number). In the present version these are not implemented, but simply perform a static sleep of 500ms. These methods should be implemented and ideally be executed on some external server in future versions of this system.

TopLink was not able to run under JBoss 4.0.5 and the planned comparison between the relation mapping services Hibernate and TopLink was thus not possible. This will probably be possible in version 5.0 of JBoss which is soon to be released. This is a very interesting comparison that could reveal inefficiencies in relation mapping services involved and will further improve this testing suite.

As mentioned above JBoss version 5.0 is soon to be released and a direct comparison between version 5.0 and 4.0.5 would also be interesting. Such a comparison could also be extended by deploying this enterprise application to other application servers with the same backend database and do a comparison between different application servers.

The maximum output achieved in this test was not more than just above 20 service interactions per second, this is a somewhat disappointing result considering the simplicity of the logic involved in the TPC-App specification. An interesting implementation would be a complete server-client implementation excluding JavaEE, EJB and Hibernate to see what performance is achieved in such a system. This could give an indication of how much overhead these services add to client-server application, and could support the following thesis: if a system is basic and not a vast number of advanced services are required JavaEE, EJB and Hibernate adds more overhead than it simplifies developers work.

The database server is disk intensive and relies heavily on disk caching to perform satisfactory. All updates of data are written to disk (in contrast to read operations that only need to access memory if a page hit occurs) and this is a costly operation. Some optimisation to this operation can be performed. When an update occurs the database engine writes the data to disk and also writes a new entry to the transaction log. This involves four hard disk operations: seek to actual data location, write new values, seek to transaction log location and write new entry. If the transaction log was stored on a different disk this could be done in parallel and only two sequential operations is needed: seek to data location, write new data. There were no time to perform tests on this optimisation in this thesis; further work conducted on this system should include testing on this optimisation as well.

7 List of abbreviations

EB	Emulated Business
EJB	Enterprise Java Beans
J2EE	Java Enterprise Edition (version 1.4)
JavaEE	Java Enterprise Edition (version 5)
JAX-RPC	Java API For XML Based RPC
JAX-WS	Java API For XML Based Web Services (JAX-RPC's successor)
JDBC	Java Database Connectivity
JMS	Java Message Service
JPA	Java Persistence API
ORM	Object Relation Mapping
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SIPS	Service Interactions Per Second
SOAP	Simple Object Access Protocol
SUT	System Under Test
TPC	Transaction Processing Performance Council
WSDL	Web Service Definition Language

8 Bibliography

Burke, B., & Monson-Haefel, R. (2006). *Enterprise JavaBeans 3.0.* O'Reilly.

Coulouris, G., Dollimore, J., & Kindberg, T. (2005). *Distributed Systems, Concepts and Design.* Addison Wesley.

Microsoft. (2006, June). *Web Services and the Microsoft Platform*. Retrieved from MSDN: http://msdn2.microsoft.com/en-us/library/aa480728.aspx

Navathe, E. (2004). *Fundamentals of Database Systems.* Addison Wesley.

Ort, E. (2004). *Ease of Development in Enterprise JavaBeans Technology.* Retrieved from http://java.sun.com/developer/technicalArticles/ebeans/ejbease/

Sun. (2007). Retrieved from Sun Developer Network: http://java.sun.com/javaee/overview/faq/persistence.jsp

Sun. (2007). *Enterprise JavaBeans*. Retrieved 2007, from Sun Developer Network: http://java.sun.com/products/ejb/

The Apache Software Foundation. (2007, January). *Apache Axis2/Java*. Retrieved from http://ws.apache.org/axis2/

The Linux Foundation. (2006). *OSDL Database Test Suite*. Retrieved from The linux Foundation: http://old.linux-foundation.org/lab_activities/kernel_testing/osdl_database_test_suite/

Transaction Processing Performance Council. (2005). *TPC-App*. Retrieved from TPC: http://www.tpc.org/tpc_app/