Cache aware load balancing for scaling of Multi-Tier Architectures

Neeraj Santosh Tickoo

Master of Science

School of Computer Science

McGill University

Montréal, Québec, Canada

January 2011

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science in Computer Science

ACKNOWLEDGEMENTS

This thesis would not have been possible without the able guidance and support of several individuals who contributed and offered their valuable assistance in the preparation and completion of this study.

First and foremost, I would like to extend my sincerest gratitude to my supervisor *Dr. Bettina Kemme* for giving me an opportunity to work on this thesis. This work would not have been possible without her ideas, opinions, guidance and financial support.

I would also like to thank the School of Computer Science Help Desk team, which helped me in setting up the cluster. Special thanks to *Andrew Bogecho* and *Ron Simpson* who helped me in the critical stages of the system set up. I would also like to thank *Kamal Zellag* and *Shamir Ali* for our insightful discussions.

Finally, I would like to thank my family, which has supported me since I came to Canada and steadily gave me moral support throughout my study.

ABSTRACT

To keep pace with the increasing user base and resulting processing requirements, enterprise and e-commerce applications need constant innovation in their application design and system architecture. Scalability and availability are the basic principles that must be adhered to by the businesses if they want to retain and expand their customer base. The most popular design which provides for both availability and scalability is when the application tier is replicated. In it, all the application servers share a single database, and to prevent the database from becoming the bottleneck in a high volume scenario, caching layers are deployed in each application server. By serving requests from the local cache instead of going to the database, response times are reduced and the load at the database is kept low. Thus, caching is a critical component of such architectures. In this thesis, we focus on object caches at the application tier, which cache Java EE entities. Our target applications are e-commerce applications which are database driven and are resource intensive.

In this thesis we design a cache aware load balancing solution which makes effective usage of the caching layer. This results in a more scalable application tier of a multi-tier architecture. Most of the load balancing solutions present in literature are cache agnostic when making the dispatching decision. Example solutions like round-robin cause duplication of the same cache content across all the application servers. In contrast, we present a cache aware load balancing algorithm, which make best possible effort to prevent the duplication of cached entries across the different caches in the cluster, enabling us to make a more efficient usage of cache space available to us. This in turn, results in less cache evictions. We also extend our cache aware load balancing algorithm to take into account the dynamic nature of the application server cluster where the nodes can come up and shutdown as the system is running. The evaluation of our implementation shows improvements in response time and throughput of a well known e-commerce benchmark compared to existing strategies.

ABRÉGÉ

Afin de suivre le rythme croissant d'utilisateurs ainsi que les demandes de traitements résultants, les applications entreprise et de commerce électronique ont besoin d'innovations régulières dans leur conception et architecture. L'extensibilité ainsi que la disponibilité sont primordiales pour tout type d'affaires ayant intérêt à garder, voir même étendre, leur clientèle. L'architecture la plus populaire qui fournit en même temps l'extensibilité et la disponibilité est celle pour laquelle le serveur d'applications est répliqué. Une architecture au niveau de laquelle les serveurs d'applications partagent une seule base de données et chacun d'entre eux utilise des couches de cache afin de réduire la charge sur la base de données. En servant les requêtes à partir du cache local, au lieu de les servir à partir de la base données, les temps de réponses sont réduits et la charge de traitement de la base de données est maintenue à un bas niveau. Ainsi, la mise en cache est une composante critique pour ce type d'architectures.

Dans cette thèse, on se concentre sur la mise en case d'objets au niveau du serveur d'applications, qui met en cache des entités Java EE. On vise principalement les applications de commerce électroniques qui sont basées sur les bases de données et qui demandent assez de ressources. Dans cette thèse, nous concevons une solution de balancement de la charge qui tient en compte la mise en cache, ce qui rend l'utilisation de la couche du cache assez effective. Ceci résulte en un serveur d'applications assez extensible pour les architectures multi-tier. La plupart des solutions de balancement de la charge ne tiennent pas en compte la mise en cache lors de la distribution de leur requêtes. Par exemple des solutions comme le round-robin entraînent la duplication du même contenu du cache à travers tous les serveurs d'applications. En revanche, nous présentons un algorithme de balancement de la charge qui tient en compte la mise en cache et qui fait de son mieux pour éviter la duplication des entrées mises en cache à travers tous les caches distribués. Ceci nous permet d'utiliser d'une façon efficace l'espace de cache disponible et de réduire le nombre d'expulsions d'entités à partir du cache. Au niveau de notre algorithme de distribution de la charge, et qui tient en compte la mise en cache,

nous prenons en considération le nombre dynamique des applications serveurs. En fait, lors de l'exécution d'un système réel, les noeuds de serveurs peuvent joindre ou quitter le système à n'importe quel moment. L'évaluation de notre implémentation montre des améliorations en terme de temps de réponse et de débit de requêtes pour un benchmark bien connu, comparativement à des stratégies existantes.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

The Internet has experienced tremendous growth since its inception. According to Internet World Stats [1], there has been a 444% increase in the number of Internet users in the world between 2000 and 2010. At the same time, e-commerce businesses, which use the Internet to carry out business activities, have experienced a phenomenal increase in user base because of ease of use and simplicity. The online sales from e-commerce websites are increasingly making a huge proportion of the revenues of the companies. A sense of this can be gauged from the fact that an e-commerce website like eBay has an active user base of more than 90 million spread all over the world [2]. According to the eBay website [2]

"*In 2009, the total worth of goods sold on eBay was $60 billion which amounts to a figure of $2,000 every second"*.

Customers expect e-commerce systems to be highly available and have acceptable quality of service. This quality of service has to be guaranteed irrespective of the increasing consumer base which puts more and more strain on the existing system resources. As is obvious from above, a few seconds of slow response time can cause a huge loss of revenue. Thus, scalable, flexible and high performing systems are needed for the survival of the e-commerce businesses, to maintain an edge over other competitors and to retain the customer base.

To this end, multi-tier architectures have emerged as the dominant distributed computing solution which assures flexibility and scalability. Such a system architecture forces the developers to develop application components targeted towards different tiers namely: client tier, web tier, application tier and database tier. In it, the client tier provides the interaction point for the end users, the web tier handles `HTTP` requests and returns `HTML` content to the end-user, the application tier hosts the logic which solves the actual business problem and the database tier stores business critical data. This separation of concerns not only simplifies system design and maintenance but also allows system administrators to keep pace with the increasing consumer load by adding additional hardware to a particular tier or replicating a particular tier. Replication if often preferred as it is more flexible and cheaper. In a replicated system, a load balancer abstracts the existence of the multiplicity of the servers at a particular tier and becomes the point of contact for all its clients. Though replication can be used for all the tiers, it is less common for the database tier as data replication is more complex. However, e-commerce applications are often data-driven and have huge dynamic content needs. Thus as the customers base increases, the database layer easily becomes the bottleneck. In such situations, system administrators usually incline towards having a caching layer at the application server level to avoid the database becoming the bottleneck and to improve response times. Having a caching subsystem increases the performance as it avoids re-computation and redundant access to the database for already cached items.

In a Java EE multi-tier architecture, all of this translates into having a cache at the application tier which caches the most frequently used database objects. Often, there is a cluster of Java EE application servers, each hosting its own object cache and all of them sharing a single database. The cluster of Java EE application servers is shielded by a load balancer which forwards the requests to them and is also responsible for replying back to the clients.

In the most common configurations, the load balancer distributes the requests to the application servers in a round robin manner. Such a load balancing approach is very flexible and simple, but it introduces the duplication of cached objects in all the caches as

the time passes by as each cache keeps the same most frequently used objects. For example in a cluster with *N* servers, each having *D* Gigabytes of cache space available, one will end up with merely more than *D* gigabytes of cached data. This prevents the system from exploiting the entire cache space available. Also, having duplicates means that we need to have mechanisms that ensure cached copies are consistent with each other. At the same time, duplication causes under-utilization of the caching component which causes the middle tier to contact the database more frequently resulting in huge database traffic and thus, preventing the scalability of the database component.

In this thesis, we work towards a *cache aware load balancing* (CALB) algorithm at the application server level, which tackles the above challenges associated with the caching solution. This is achieved by having a smarter load balancer which distributes each request to the application server which has already cached the data needed for this request. This work is inspired by Elnikety *et al.* [3] However, in Elnikety *et al.* [3], the load-balancer was in front of a replicated database system, while this thesis works at the application server level.

In our work, we concentrate mainly on applications which are database driven and have dynamic content needs (e.g. e-commerce websites). For such systems our cache-aware load balancer works at the application server level, distributing requests to a cluster of nodes in such a way that it tries to partition the content into nearly disjoint subsets which are then cached by the different nodes in the cluster. The load balancer also takes into account the cache space available on each server machine. This consideration prevents allocating too many requests to the same application server, preventing overfilling of the cache at a particular application server and thus, avoiding cache evictions. All of this ensures that we achieve a high cache hit rate, which in turn translates into less network traffic directed towards the database layer and lower response time.

We also adapt our cache aware load balancer to a set-up where we have a dynamic number of nodes in the cluster. Commercial load balancers like Linux Virtual Server [4], Apache httpd load balancer [5], do not handle the addition or removal of a node when the

system is running. In contrast, our load-balancer can handle the addition and removal of nodes while the system is running and at the same time retains the features of our earlier implementation.

More precisely, the contributions of this thesis are:

- We first design the CALB algorithm for a cluster of application servers. In the initial design, the number of nodes in the cluster is fixed and does not change while the system is running.
- We then extend the CALB algorithm so that it can handle the addition and removal of application servers in the cluster while the system is running.
- We then evaluate the performance of CALB and compare it against other load balancing algorithms.

Thesis Outline

The remainder of this thesis is structured as follows:

In Chapter 2 we discuss the background information on multi-tier architectures, Java EE technologies, scalability in multi-tier architectures, caching and load balancing. We also discuss different types of load balancing approaches. We then focus on content-aware load balancing algorithms and discuss the related work in that field.

In Chapter 3 we describe the contribution of this thesis in detail. We touch upon the design and implementation details.

In Chapter 4 we discuss the implementation and provide performance results for a common multi-tier benchmark. We also compare the performance of our solution with other known content-aware and content-blind load balancing algorithms.

Finally in Chapter 5 we draw conclusions and discuss future work.

Chapter 2

Background

The emergence of the Internet led to the adoption of the client/server architecture for the design and deployment of distributed applications. Gradually, the client/server architecture became the de-facto model for distributing a system over a network. The client/server architecture divides the application into the two layers of clients and servers. The client tier is responsible for presenting an interface to the end user which it uses for interacting with the system. The client tier may also host some processing logic depending on the way the application has been designed. On the other hand, the server layer is responsible for hosting the business logic necessary for satisfying the client requests. The server is responsible for executing the bulk of the processing. The client server interaction is always started by the client and follows a request/reply pattern. A typical example of the client server architecture is the World Wide Web. A web server hosting some website is an example of a server and any computer requesting the web content hosted by this web server is the client.

The biggest advantage of the client/server architecture is the separation of concerns by distributing the logic over different computers in the system. This also allows greater ease with application maintenance.

The client/server architecture is also referred to as two tier architecture. A client/server system can be shown as in Figure 2-1.

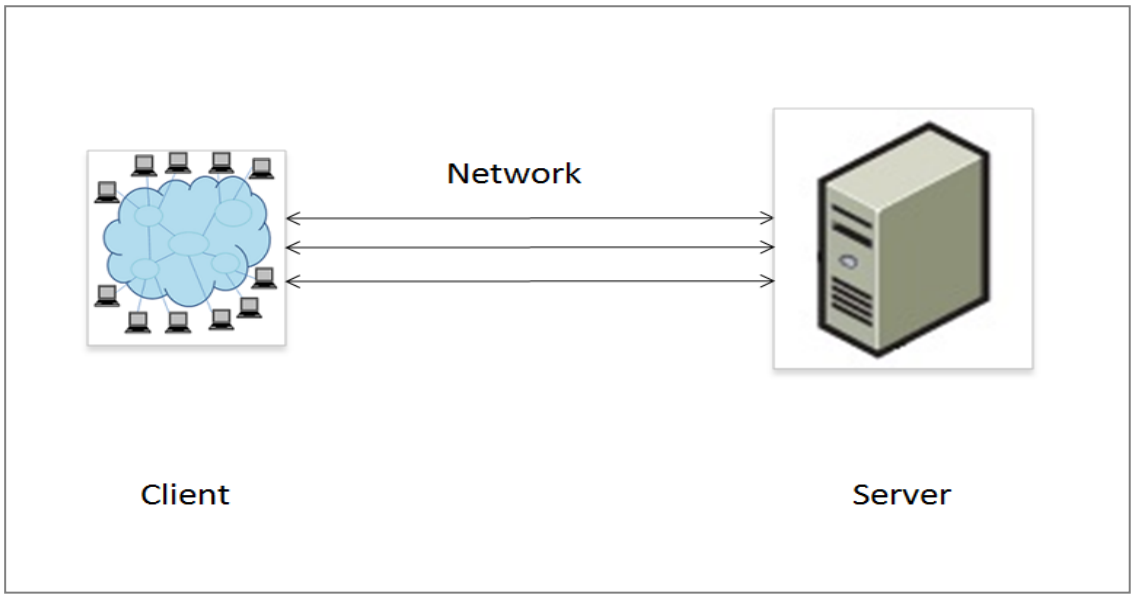

Figure 2-1 - Client/Server Architecture

The tremendous growth of the Internet in the last few decades demanded a more scalable and flexible distributed architecture for applications. A simple client/server architecture has scalability issues as the single server quickly becomes the bottleneck when the user base increases. This led to the emergence of multi-tier architecture.

2.1 Multi-Tier Architecture

In a multi-tier architecture, an application is designed and distributed across different tiers where each tier is responsible for providing a dedicated functionality. In such an architecture design, a change in one layer/tier does not affect the other layers. Therefore, this architecture design pattern provides distinct advantages such as:

- Reusability
- Flexibility

- Improved Security
- Maintainability
- Scalability

A 3-tier architecture is the most common instance of a multi-tier architecture. A 3-tier architecture consists of the following layers:

1. **Presentation/Client Tier**: It is at this tier, where the interaction with the end user takes place.

2. **Middle/Application Tier**: The middle tier hosts the complex business logic and it is here where the bulk of the processing and computations takes place.

3. **Database Tier**: The database tier consists of a database and is responsible for storing business critical data.

A pictorial representation of the 3 tier architecture is show in Figure 2-2.

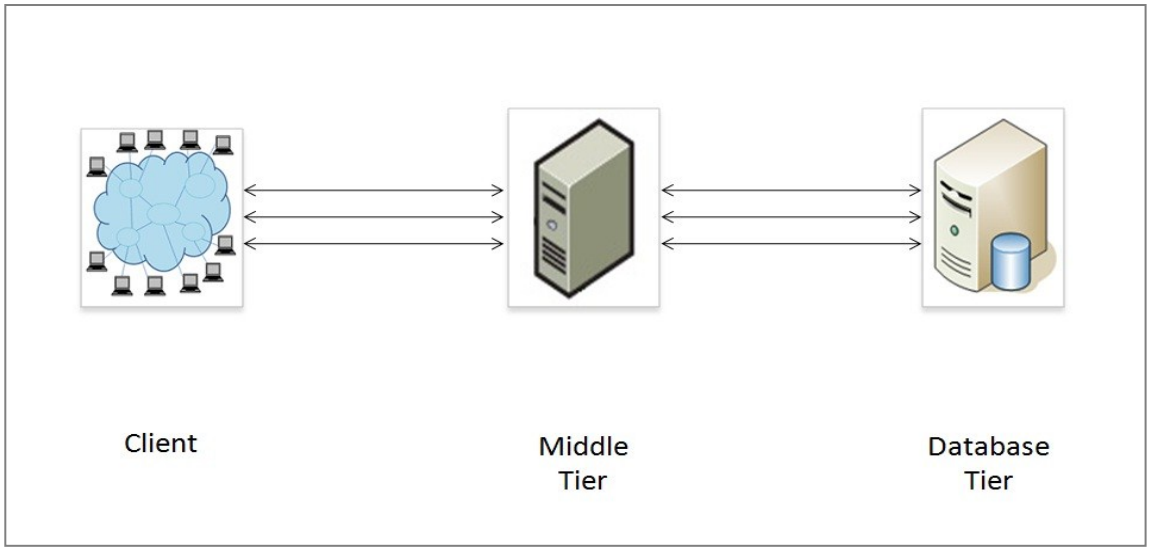

Figure 2-2 - 3-Tier Architecture

2.1.1 Java Enterprise Edition (Java EE) Architecture

As the use of the Internet has grown for carrying out e-business activities, there has been a strong need to have a framework which developers can use to quickly develop distributed and reliable multi-tier applications. The Java Platform, Enterprise Edition or Java EE [6] provides such a specification for developing distributed, reliable multi-tier modular applications.

Sun Microsystems, now owned by Oracle, released the Java Platform Enterprise Edition– Java EE (previously known as J2EE), a specification for the design and development of distributed enterprise applications.

The Java EE has been described on its Oracle documentation homepage [6] as follows:

"The Java EE application model defines an architecture for implementing services as multi-tier applications that deliver the scalability, accessibility, and manageability needed by enterprise-level applications. This model partitions the work needed to implement a multitier service into the following parts:

- *The business and presentation logic to be implemented by the developer*
- *The standard system services provided by the Java EE platform*

The developer can rely on the platform to provide solutions for the hard systems-level problems of developing a multitier service."

Since Java EE is a specification, different vendors provide their implementations for it. The Java EE specification includes API's for JDBC, RMI, e-mail, web services, etc [7]. A vendor whose application server conforms to these specifications is known as Java EE compliant. Some examples of open-source application servers that follow the Java EE specification are JBoss [8], JOnAS [9] and Glassfish [10] etc. A Java EE compliant application server provides an environment where a developer can leverage the existing features like transaction management, concurrency control, security, etc and concentrate on developing the business logic of the problem at hand rather than focussing on re-

inventing the wheel every time an enterprise application needs to be developed. This leads to a shorter development time and better application maintenance.

Java EE has been described as a 4 tier architecture in [6] consisting of the following 4 tiers (Figure 2-3) which we discuss in the coming sections:

- Client-tier
- Web-tier
- Business-tier
- Enterprise information system (EIS) tier.

In [6], Java EE applications are also referred to as 3-tier applications as they are distributed over three separate locations, namely the client machine, the application server and the database server. In this configuration, the application server hosts both the web-tier and the business tier.

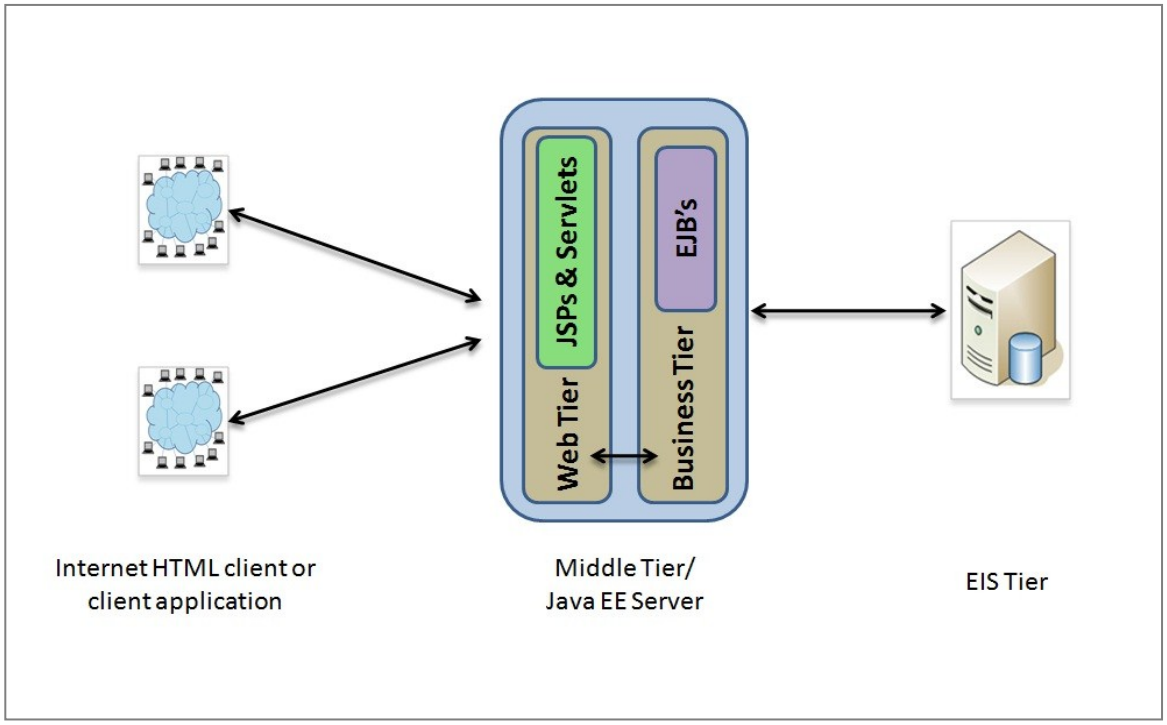

Figure 2-3 - Java EE architecture

2.1.1.1 Client Tier

The client tier is responsible for taking the input from the user, communicating with the server using an appropriate protocol (usually HTTP), capturing the response from the server and displaying the output to the end user. In Java EE, the client tier can be a web client or an application client.

A web client is also known as a thin client. The most common instance of such clients is a web browser. Such clients usually use HTML to display the interface to the end user. In such thin clients the bulk of the processing is done on the server side and the client does not contain any business logic.

Application clients are usually used when users need a richer graphical user interface as that provided by a web client.

2.1.1.2 Web Tier

In the Java EE architecture, it is the web tier which intercepts the requests and is responsible for generating the HTML pages which are then displayed by the client tier. The web tier is associated with a web server and its major components are JSPs and Servlets. JSPs (also known as JavaServer Pages) and Servlets are Java technologies which simplify the generation of dynamic content. Static HTML pages are not considered as web components by the Java EE specification [6].

2.1.1.3 Business Tier

This tier hosts the business logic needed for solving business problems concerning a particular domain like an e-commerce website, financial industry etc. The Java EE

specification provides the concept of Enterprise Java Beans (EJB) [11] towards this end. EJBs are components that are responsible for handling the requests of the clients, processing them, interacting with the database and generating the results. The results are then usually transformed into an `HTML` response and returned to the client by the web tier. The business tier consists of an Enterprise Java Bean server which hosts the EJBs.

As mentioned in [6], Java EE defines 2 types of EJBs namely:

- Session beans
- Message-driven beans

J2EE 1.4 [7], the predecessor version of Java EE 5, had an additional type of EJB called Entity Bean. Since the release of EJB 3.0 with Java EE 5, Entity Beans have been replaced by the Java Persistence API [12] also known as JPA which principally deals with persistence entities.

Here, we mainly discuss JPA/Java Persistence Entities as they are most relevant in the context of our thesis. We later touch upon session beans and message-driven beans for the sake of completeness.

JPA/Java Persistence Entities

Due to some shortcomings in the Entity Bean's persistence model used in J2EE 1.4 and its predecessor versions, the new EJB 3.0, defines a completely new persistence framework. The new specification is called the Java Persistence API.

The Java Persistence API defines a framework for handling relational data stored in a database in a Java application. With this, the programmer receives an object-oriented view on the relational data stored in the database. The programmer simply works with Java objects also called entities. Handling the Java objects through the JPA framework enables the programmer to persist the object state automatically without having to specifically code for it, enabling the Java objects (entities) to outlive the lifetime of the application.

Entities in JPA are the object representation of a domain object. Typically, an entity class represents a table in the database. An instance of that entityclass then represents a row in the database table. This mapping from a class/entity to a database table is accomplished with a Object Relational Mapping (ORM) facility. The ORM tools use annotations or XML configurations supplied by the developer to map the entities and different relationships to the tables stored in the database. Figure 2-4 shows an example of this mapping. In the example, we have a *User* entity which is the class representation of the *users* table in the database.

Examples of ORM frameworks widely used are Hibernate and TopLink.

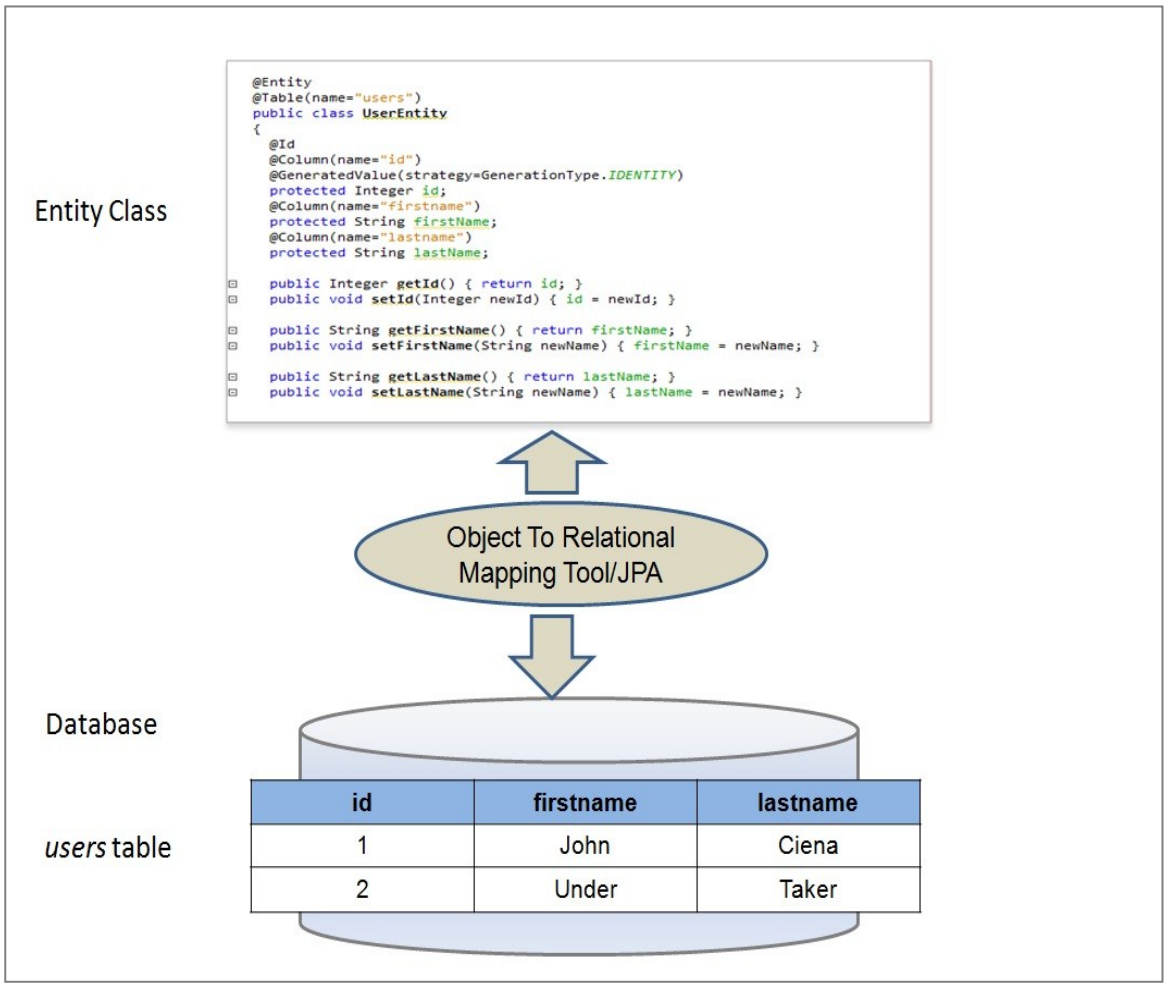

Figure 2-4 - Object to Relational Mapping

Session beans

In [6], session beans are described as non persistent components of the Enterprise Java Bean server. They encapsulate business logic. Such session beans expose methods which are then called by client to perform a particular operation (e.g., add an item to a shopping-cart). A session bean models a particular domain process. Session beans are of two types. Stateless beans do not keep any state between method invocations while stateful beans maintain transient data and are usually associated with a particular client session.

Message-driven beans

In [6], message driven beans have been described as

"A message-driven bean is an enterprise bean that allows J2EE applications to process messages asynchronously. The messages can be sent by any Java EE component or by a JMS application or system that does not use Java EE technology."

2.2 Scalability

With the constant increase of the user base, the number of user requests that must be handled by any of the tiers increases tremendously. Thus, any tier faces the challenge to scale in order to accommodate the increasing workload (scalability) and remain available at all times (availability).

Methods to scale a tier of a multi-tier architecture usually fall into two broad categories:

- **Vertical Scaling (scale up):** Vertical scaling adds resources to a particular node of a tier. This typically translates in the addition of more CPUs or additional memory to a single machine [13] as shown in Figure 2-5.

- **Horizontal Scaling (scale out)**: In Horizontal scaling (Figure 2-6), multiple hardware or software systems are deployed at the same tier and the system is configured in such a way that they operate as a single unit. For example, in the case of application servers or database servers, horizontal scaling translates into having multiple application servers or database servers satisfying the user requests. Horizontal scaling typically employs the techniques of clustering (running a set of servers in a cluster) and load balancing. In this design, all servers share the load. If one server fails then the load of that server is distributed to the rest of the machines in that tier.

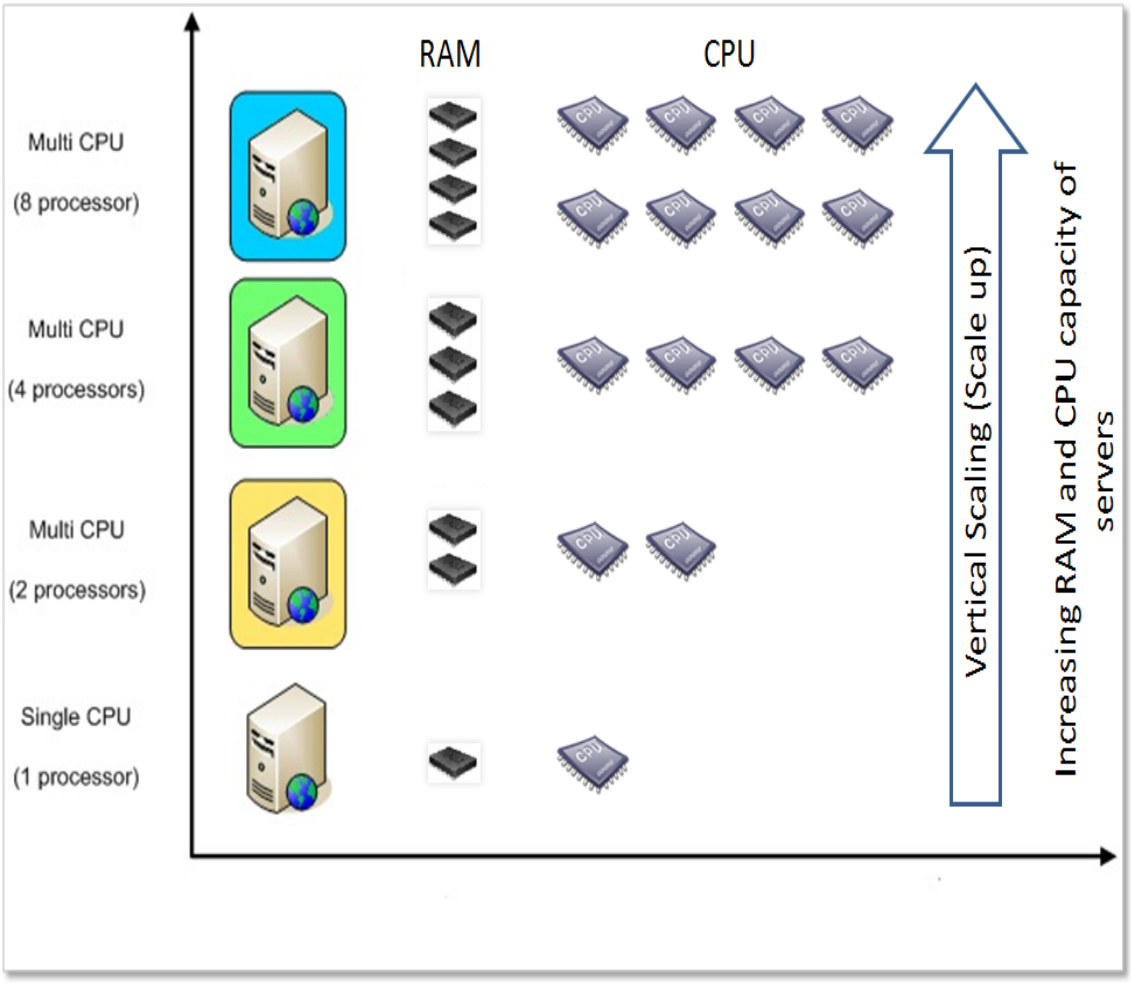

Figure 2-5 - Vertical Scaling

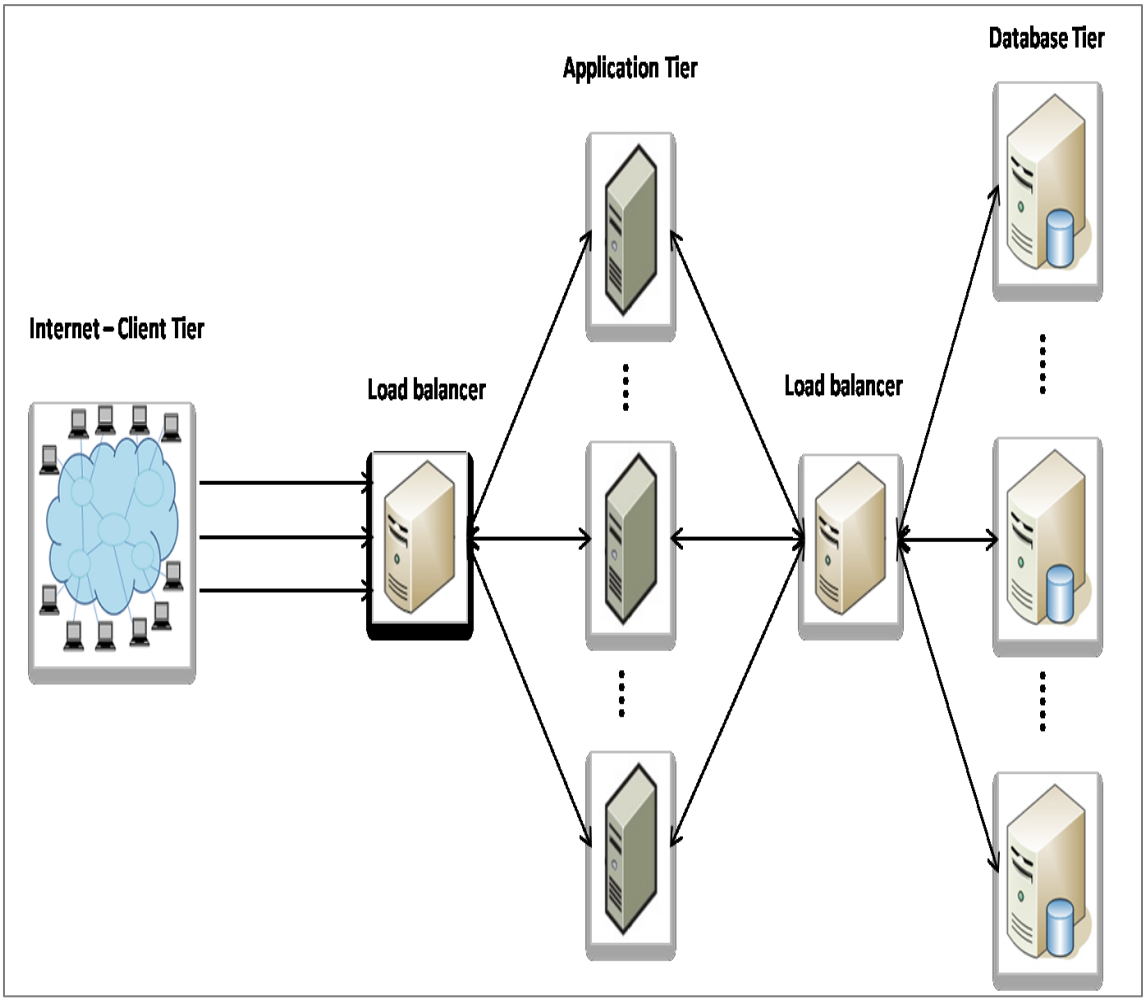

Figure 2-6 - Horizontal Scaling of the application tier and the database tier

Both the discussed approaches have certain trade-offs as listed below:

Vertical Scaling:

Cons:

- Expensive: Adding hardware to an existing system to scale the tier is a very expensive approach.

- Single point of failure. Vertical scaling puts all the resources in a single machine. If that machine goes down, the entire system will be unavailable. This means, vertical scaling addresses only scalability but does not address availability.

Pros:

- Vertical scaling is easy to implement and hardware upgrades usually do not involve changes at the application level.
- Vertical scaling has low administration overhead as one has to manage a single machine.
- Vertical scaling has a single machine in each tier. This makes the application design simpler as one does not need to take into account the distribution principles while designing the application.

Horizontal Scaling Pros and Cons:

Cons:

- With horizontal scaling, many machines comprise a particular tier. This means increased management complexity and administration overhead.
- Using horizontal scaling, the application design follows a more complex distributed programming model.

Pros:

- Horizontal scaling is cheaper than vertical scaling as horizontal scaling is usually achieved by adding cheap commodity machines to a particular tier.
- Horizontally scaled systems can handle failures better as a single node failure does not cause the entire application to go offline. This enhances the system's fault tolerance and stability.

Due to the desirable features of horizontal scaling, it is the more common approach adopted to scale the web and application tier. But horizontal scaling causes the introduction of multiple servers. It would be cumbersome for the client to be aware of this distribution. For example, in a web-based architecture, to adapt to this distribution, the client can manually select alternative URL's each representing a web server that hosts the website. But with this approach user transparency is lost.

In order to abstract from this multiplicity, usually a load balancer is placed in front of a particular tier and it becomes the point of contact for the preceding tier. This means, the load balancer intercepts the requests of the tier it is abstracting and then is responsible for forwarding the requests to one of the servers of the next tier and returning the reply to the client. As shown in Figure 2-6, the clients send their web requests to the load balancer sitting in-front of the application server cluster, which then forwards the requests to one of the servers. Similarly, a load-balancer in front of the database tier distributes requests to the database replicas. Commercial systems, however, seldomly use horizontal scaling for the database servers. The problem is that the data needs to be replicated and maintaining data consistency is challenging.

2.3 Load Balancing

Load balancing is the process by which the requests are distributed across multiple servers at a particular tier. Cardellini *et al.* [14] propose a classification of the load balancing approaches for the web-server tier at the following four different levels:

1. **Client Based Approach**: In this approach, the client is aware of the replicated nature of the servers at a particular tier. The client selects a node and directly communicates with it by sending requests to it. The selected server is then responsible for generating the reply and responding back to the client. The client based approach has limited practical applicability and is not scalable as the client has to know all the server replicas.

2. **DNS Based Approach**: A DNS server maps an IP address to a domain name. In the DNS based load balancing approach, the requests are forwarded to different servers by mapping the same domain name to different IP addresses in the DNS server. Thus, when a DNS server receives a DNS request for resolving a domain name, it maps it to a particular IP address based on the configured load balancing policy like round robin scheduling. After that, the request is dispatched to one of the servers in the group.

3. **Dispatcher Based Approach**: In a dispatcher based approach, the dispatcher receives all requests from clients and distributes the requests to servers. Here, the dispatcher acts as the central point of contact for all the clients. Routing the requests to different servers is done transparently. Cardellini *et al.* [14] also describe several variations of dispatcher-based architectures using different routing mechanisms namely:
   - Packet rewriting,
   - Packet forwarding, and
   - Request redirection.

   Examples of request distribution algorithms at the dispatcher are round robin scheduling, least-connection and weighted round robin scheduling. The mentioned algorithms are very simple in nature. In principle, however, dispatcher based approaches can be more sophisticated and can implement a whole range of distribution policies. As compared to the other approaches, dispatcher-based approaches are more flexible and give more control over the request distribution policies.

4. **Server Based Approach**: In this load balancing approach, the dispatching of the requests takes place in two stages. In the first stage, a server is chosen using the DNS based approach. In the second stage, the chosen server then reassigns the request to any other server in that tier. This approach allows all the servers to participate in the load balancing decision.

In this thesis, we only consider dispatcher-based load balancing approaches.

Most load balancing algorithms fall into the below two categories

- Content blind load balancing algorithms.
- Content aware load balancing algorithms.

2.3.1 Content Blind Load Balancing

Content blind load balancing algorithms make the dispatching decision without taking into consideration the information contained in the request sent by the client. However, the dispatcher can take system's state into account, e.g. the server CPU load or the number of open TCP connections. Some examples of content blind load-balancing policies are:

- **Round Robin**: In round robin scheduling the load balancer dispatches the requests to the servers one by one. For example: if we have 3 servers A, B and C and the clients send 5 requests I, II, III, IV, V in the said order, then the load balancer will send request I to server A, request II to server B, request III to server C, request IV to server A and request V to server B.

  As seen from the above example, the round robin scheduling policy does an equitable distribution of requests to all the servers over time.

  However, this load balancing policy does not take into account the capacity of different servers in terms of their processing capability, available RAM etc while making the dispatching decision.

- **Weighted Round Robin**: Round robin load balancing policy is best suited for an environment where all the servers are homogenous. To overcome this limitation, weighted round robin scheduling assigns weights to different servers and then allocates requests to them according to that assigned value. The weights can

reflect the processing capacity of one server as compared to the other. For example if we have two servers A and B, and server A has weight 2 and server B has weight 1, then Weighted Round Robin will assign twice as many requests to server A as compared to server B.

- **Least Connection**: In this load balancing policy, the load balancer keeps a track of open connections at each server. It then assigns the incoming request to the server with smallest number of open connections.

- **Weighted Least-Connection**: This load balancing policy is similar to Least Connection but weights are assigned to different servers in the cluster and the request distribution is done in accordance to those values.

- **Least Loaded**: This load balancing policy needs some metric which reflects the load on a particular server. The metric can be the CPU load, free memory etc on a machine. The requests are then allocated to the server which has the lowest value in the load metric. In this load balancing algorithm, the server machines have to periodically send the load metric values to the load balancer. This can be implemented via daemon processes.

- **Random Server**: In this load balancing policy the requests are assigned to the servers randomly but care is taken to ensure that there is equitable distribution of requests to each server.

2.3.2 Content Aware Load balancing

Unlike the content-blind load balancing, content aware load balancing algorithms utilize the information contained in the requests from the clients to direct the traffic to a particular server replica. Such a load balancer works at the application layer. An

interesting property of content aware load balancing algorithms is that they can be made application specific. However, such load balancing policies are more complex to implement as compared to content-blind load balancing techniques.

In this thesis, we concentrate on content aware load balancing algorithms. Work related to Content aware load balancing algorithms will be explained in Section 2.5.

2.4 Caching

In multi-tier architectures, as demand for services continue to grow at a rapid pace, it is usually the application tier which is scaled horizontally. Horizontal scaling adds a lot of processing power to the application tier. The application tier in turn depends heavily on the database layer. As disk access is always slower than main memory access, and due to the data intensive nature of applications likes e-commerce systems, all of this means that the database easily becomes the bottleneck in the system.

So, in addition to scaling the application tier horizontally, other steps need to be taken to increase the performance of the database component and thus, scale the entire system.

As discussed before, the database tier here can be scaled either horizontally or vertically. However, vertical scaling is expensive and horizontal scaling is complex as it requires some replica control protocols to keep the database replicas consistent. Therefore, an alternative solution is to not scale the database but to keep the load that is submitted to the database low. This can be achieved by providing a caching layer between the application server and the database.

The caching solution is one of the most common and flexible approaches used to keep the load on the database tier low. In this thesis, we concentrate on caching to improve the scalability of the multi-tiered architecture.

A cache has been defined in [15] as a temporary data store which either duplicates data located elsewhere or stores data which is the result of some computation. A cache component results in superior performance if the requested data can be satisfied from the results or data stored in the cache. If the requested data is found in the cache it is known as a *cache hit*. Otherwise, if the requested data is not found in the cache it is called a *cache miss*. In case of a cache miss, the data requested is recomputed (e.g. retrieved from the database), stored in the cache and then returned to the client. This constitutes a *cache put*. The ratio of the number of requests satisfied from the cache to the total number of requests received is known as *cache hit rate*. A high cache hit rate is desirable as that means many of the requests are satisfied from the cache resulting in better performance. An element stored in the cache is also associated with a Time to Live (TTL). When TTL for an element elapses, it is said to have *expired* and is evicted from the cache.

In a multi-tier architecture, a cache layer can be placed on the

1. Database tier,
2. Application tier,
3. On both database tier and application tier
4. On a machine between database tier and application tier.

In this thesis, we look at a configuration where each application server in the cluster hosts a caching component (2^{nd} approach). Placing the caching component at the application tier significantly improves the application performance due to the following reasons:

1. There is no network latency for the requests satisfied from the cache as there are no network round trips to the database.
2. There is no marshalling and un-marshalling processing cost.
3. Less time is spent on creating and destroying the network connections between the database and the application server.

4. Less network traffic is directed towards the database tier. This results in lower load on the database tier. Therefore, those requests that are submitted to the database can be served faster.

All of this results in the improvement of the following performance metrics:

1. Response time experienced
2. The client and the overall throughput that the system can achieve.

Benefits of having a caching subsystem at the application tier was also demonstrated by Luo *et al* [16] where they develop a research prototype called DBCache at the application server level for alleviating the load on the database server.

2.4.1 Challenges of caching in horizontally scaled multi-tier architectures

Current application server clusters that use caching often have the following configuration. There is a cluster of application servers all of which have their own cache. A load balancer is placed in front of the application server cluster, and it usually uses a content-blind load balancing algorithm like round robin for dispatching requests to the application servers. Due to the nature of the round robin algorithm, the requests will be forwarded to the application servers one by one and we would expect that after a certain period of time, all caches host copies of the same objects, namely the most popular objects. In other words, the most popular objects would be replicated across all the caches.

A big disadvantage of this configuration is that we are not able to exploit the true cache capacity available to us which is the sum of the cache capacities of all the nodes. Assume a configuration with N servers, and each node has D Gigabytes of cache, in the above set up, we end up with barely more than D Gigabytes of total cached data in the entire cluster. Every server just has the copies of the same items.

This duplication of cached data not only prevents the scaling of the caching layer, which in turn prevents the scaling of the database layer, it also leads to inconsistency issues as the update in one copy is not reflected in other copies. This can cause different clients to see different versions of the same data item, which is clearly undesirable.

Thus, in summary, the main problem of the caching approach with a naive load balancing policy at the load balancer is:

- The lack of exploiting the available cache space.
- Maintaining consistency between cached copies.

These problems are some of the road blocks of deploying a scalable caching layer in a clustered environment.

The above problems can be avoided by one of the following two mechanisms:

- There is a cooperation mechanism between the caches, so that an object is cached at one location and the application layer using the cache has a single view of the entire cache available in the cluster.
- There is a cache/content aware load balancing algorithm, which partitions the content among the cluster node caches. Each request is forwarded to the server whose cache hosts the partition accessed by the request. Therefore, there is minimal content replication among caches.

In this thesis, we will concentrate on the cache/content aware load balancing algorithm which tries to circumvent the problems discussed above.

2.4.2 Caching Architecture in Java EE

For applications which need or will benefit from caching, Java EE framework provides caching services. Java EE framework provides caching services at different levels namely:

1. Level 1 Cache (L1 Cache)
2. Level 2 Cache (L2 Cache)

As discussed in Section 2.1.1.3, Java EE framework offers *Entities* whose instances map to a database table row. These instances are managed by an *EntityManager*. In Java EE, the mentioned levels of caching are handled in the context of an *EntityManager* and *persistence context*. In [6] *EntityManager* and *persistence context* are discussed as:

"Each EntityManager instance is associated with a persistence context: a set of managed entity instances that exist in a particular data store. A persistence context defines the scope under which particular entity instances are created, persisted, and removed. The EntityManager interface defines the methods that are used to interact with the persistence context."

Level 1 Cache

This level of cache is always turned on by default. It caches instances within an *EntityManager*. In it, the instances are cached on per request (session) basis. Level 1 cache reduces the number of database accesses within a single transaction as multiple requests for the same objects are satisfied from the level 1 cache.

Level 2 Cache

Level 2 Cache is shared across different *EntityManager* instances. It is also usually called as a shared cache as it caches entity instances across requests or sessions. This cache needs to be enabled through a configuration property. In a typical request execution, if an entity is not found in the L1 cache, it is searched for in the L2 cache. If present it is returned and stored in L1 cache, else the results are fetched from the database and cached in both L1 and L2 cache. In this thesis we focus on L2 cache.

Level 1 and Level 2 caches are also shown in Figure 2-7.

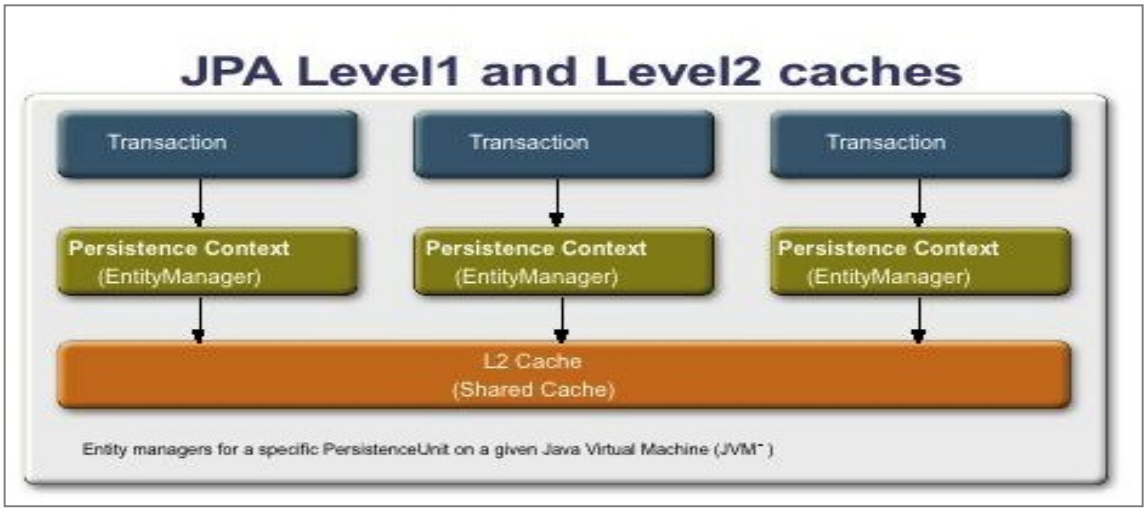

Figure 2-7 – Java EE Caching Architecture [38]

2.5 Related Work

Several solutions have been proposed as distribution policies in a content aware load balancing design. The aim of these policies is to exploit cache locality by sending the requests that ask for a specific web page to a server that is likely to have it in its cache component. By doing this, the content is divided into disjoint subsets which are then cached by various nodes in the cluster. This allows for the utilization of the entire cache space available, which in turn increases the performance of the application by increasing the cache hit rate. At the same time, such an approach also prevents duplication of cached objects and therefore avoids inconsistency issues.

As stated in [17], content-aware load balancing solutions improve the web application performance by improving the cache hit rate. Most of the content aware load-balancing policies presented in the literature have been designed for web systems hosting static web pages [17]. The central theme of such content-aware policies is to divide the content among the nodes of the cluster and then forward a request to the node which is

responsible for the requested content. Such a policy was initially proposed by Pai *et al* [18] in their locality aware request distribution strategy named LARD. LARD is a content-aware load balancing policy that considers locality of reference of static web pages while performing load balancing. In LARD, the requests for the same web objects are sent to the same node. Here, web objects are static `HTML` files, images, audio files, video files etc. By directing the web request to the same server, the requested web object is more likely to be found in the cache of the server node. LARD also monitors the load on the different servers in the cluster. This is done to avoid flooding an already overloaded server with additional requests. In LARD when the server load goes above a certain threshold, its requests are then assigned to the least loaded server in the cluster.

The concept of LARD can be explained with Figure 2-8 shown below:

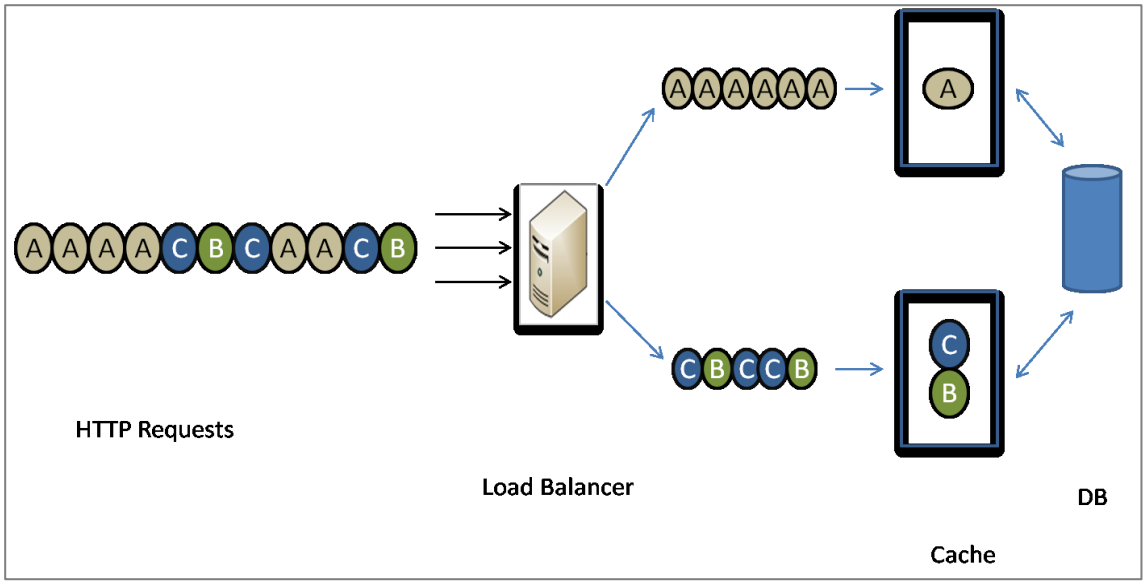

Figure 2-8 - LARD

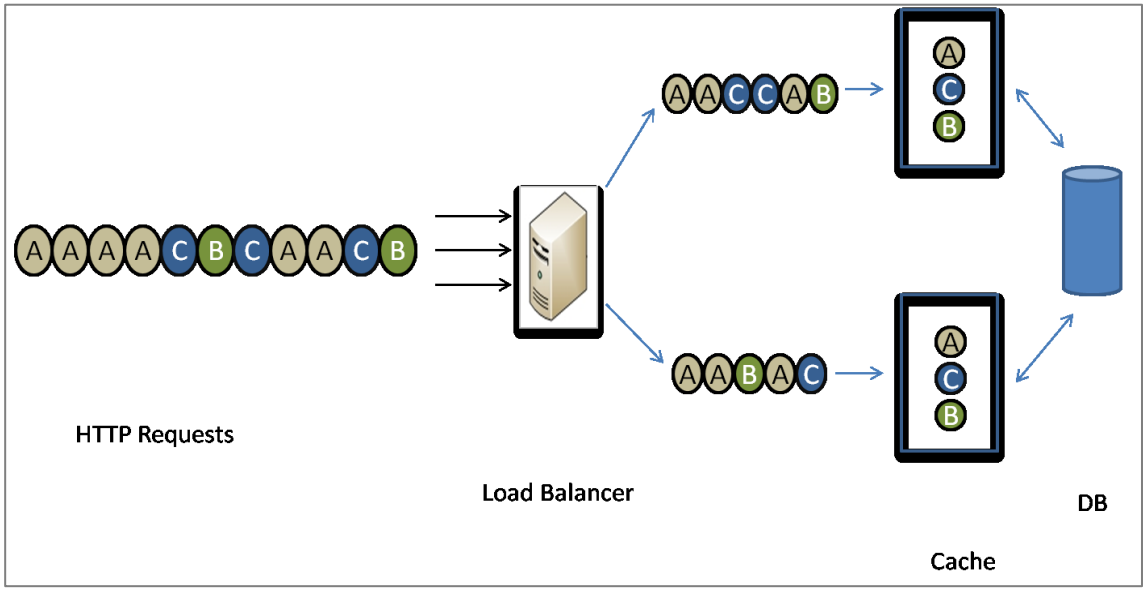

Figure 2-9 – Content Blind Load Balancing – Round Robin Scheduling

Figure 2-9 shows how a round robin load balancing algorithm (content blind scheduling) would distribute the same load. As we can see, in the round robin scheduling, the contents are duplicated in the caches of both the web servers. Supposedly, if the cache size was smaller than the combined size of all objects then it would have lead to cache evictions, resulting in worse response time.

Other work related to content aware load balancing has been presented in FLEX [19] and in HACC [20]. The work presented in FLEX [19], targets web content hosting services. FLEX is an adaptive load balancing solution which forms groups of different websites based on their memory and access rates. The access rates and the memory needs of a particular website are estimated by parsing web server log files. Based on this, the websites are then assigned to the available nodes in the system.

HACC, also known as *Harvard Array of Clustered Computers,* enhances the performance of a web server cluster which hosts both static files and acts as a document

store. It partitions the content over the nodes in the cluster and keeps on reducing the working set handled by the nodes when a new node is introduced into the cluster making the load balancing dynamic.

FLEX provides a locality aware solution for the design and management of a web hosting service whereas HACC targets clustered web application servers. Also, FLEX is based on a DNS infrastructure that allocates different websites to the nodes of the cluster.

In later work [21], the authors of FLEX propose WARD, a locality-aware distribution policy where the files that are most frequently accessed, are replicated across all the nodes in the cluster. The rest of content is divided among all the nodes with no replication. This design introduces replication for *hot* objects.

Elnikety *et al.* [3] extend the previous work in this research area by developing a memory aware load balancing (MALB) algorithm used for a cluster of replicated database servers. The MALB algorithm works by grouping transactions and dispatching them to server groups. As explained in [3], the improvement in performance is because the transaction groups formed execute in the available memory of the machine thereby reducing disk accesses. In the evaluation, the authors show that MALB greatly improves performance over other load balancing techniques – such as round robin, least connections, and locality-aware request distribution (LARD) – that do not use explicit information on how transactions use memory. Unlike the above mentioned content aware load balancing algorithms like LARD, FLEX etc which target static content, MALB targets database workloads resulting from database driven websites like e-commerce applications which need dynamic content.

Chapter 3

Cache Aware Load Balancing

In this thesis we work towards a load balancing solution which optimizes the object distribution among level 2 caches in a cluster of Java EE application servers, thus making the load balancer cache aware. In a Java EE environment with hibernate being the ORM mapping framework, level 2 cache (discussed in Chapter 2) acts as a cache store of objects across sessions. This cache remains alive till the application is running. On the other hand, there is one more cache scheme that exists which is called as level 1 cache. Usually this acts as a cache store on a per transaction basis. The objects cached here are de-allocated after each session execution.

The CALB approach discussed here is for a multi-tier architecture where we have a cluster of Java EE application servers all of which share a single database. An application server provides a managed environment for hosting enterprise applications. Such applications are developed in an object oriented environment. They usually make use of an ORM framework which converts the database table rows into objects. For the reasons discussed in the background chapter, in our configuration each application server also has its own cache. The cache is also referred to as an object cache and acts as a store of frequently accessed database items.

The enterprise applications which interest us are e-commerce applications as they are database driven, have dynamic content needs and involve transactions at the application

server level. Examples of e-commerce applications are popular websites such as eBay, Amazon, Kayak etc. For the purpose of simplicity, we shall be focussing on read-only workloads of the e-commerce applications for the purpose of evaluation.

The load balancing solution presented here is at the application server level. In particular, it is designed for the second level cache of Java EE application servers. This work is inspired by the MALB paper [3] which handles load-distribution for a replicated database cluster. Using MALB, the load balancer utilizes the working set information of the transactions to group them and then assign the groups to a database replica depending on its available memory capacity. This grouping of transactions ensures that the working set of a transaction group can be accommodated in the main memory of the database replica, thereby avoiding memory contention.

In this thesis we initially develop the CALB solution at the application server level that follows the MALB approach. In the first step we assume a static configuration with a fixed number of servers. From there, we extend the system for an environment where the application server nodes can join or leave the cluster at any time and the load balancer dynamically receives the configuration information about the application servers. Our dynamic load-balancer can handle events such as node start-up, shutdown and application deploy, un-deploy.

3.1 Fixed Cluster

We first outline the basic requirements of the CALB. In the following sections we give a description of each requirement.

To have a CALB solution, which tries to dispatch requests with the final goal of maximizing cache hits and minimize cache evictions we need the following:

1. We have to identify different types of user requests at the load balancer.
2. We have to identify the size and content of the working sets of user requests.

3. We have to group requests which have similar entity/table access pattern, taking into account the cache space available on each application server.
4. We need a dispatching mechanism that forwards requests of a transaction group to the appropriate application server.

3.1.1 Identifying different types of user requests

User interaction with an application server is usually accomplished using the `HTTP` protocol. The web server component of an application server intercepts the `HTTP` requests and then interacts with the application server. The user request could either correspond to a static web object or might request a dynamic web page. Content of a particular static web page is the same for all requests. In contrast, content for the later type of request is generated dynamically and could vary from request to request. The content generation process for a dynamic web page often involves interaction with a database. In this thesis, we are primarily interested in such user requests for dynamic content. Generating dynamic content is both CPU and I/O intensive. This is also demonstrated by Iyengar *et al* [22]. In their work they analyze the web server performance under different workloads and observe that generation of dynamic `HTML` pages is the most resource demanding part of the content generation process. Thus, properly caching such content will result in substantial performance improvement in terms of response time and throughput.

Most of the enterprise and e-commerce applications are developed with a transaction oriented scheme. Transactions guarantee atomic execution of requests. In such applications, each web request translates into the execution of a particular transaction at the application server level. A transaction might eventually issue multiple SQL statements to fetch data from the database according to the user input. This data constitutes the dynamic content of the user request. For example in eBay, requests such as *view item, display categories, display hot deals etc,* result in the execution of different

corresponding transaction types at the application server. All of the above also means that such applications have a fixed set of transaction types which can be invoked through a fixed set of user requests.

Our cache aware load balancer needs some information about the type of requests it is handling. To identify the type of a user request, in our evaluation, the client embeds the transaction type in the query string of the URL of `HTTP` request. For example, if the user wishes to view all items the client can embed an identifier "viewItems" for this request in the query string of the URL as:

> http://domainName/path/viewallitems.html?*transactionType=viewItems*

Thus, when the load balancer receives this request, it can easily figure out which transaction/request it is handling by parsing the URL.

3.1.2 Identifying the working sets of user requests

Having identified the different types of transactions/requests handled by the system, we now need to determine the data each transaction type accesses. This data is called the transaction working set. Furthermore, we have to estimate the size of the working set.

In the e-commerce applications the working set of a transaction is dominated by the entities it references. As described in the background chapter, entities map to tables in the database. The mapping is done using an ORM framework. Transactions at the application server level are then written in terms of the entities rather than tables.

When caching is involved, the entities mentioned here are cached in the second level cache. In a Java EE cache, each entity class has its own cache region. Therefore, different tables or entity classes are cached in separate cache regions.

Calculating the working set size of a transaction requires the calculation of the space the objects require in the second level cache. As for a given transaction type it is not possible to determine the specific entities accessed but only the entity classes (i.e. tables), the working set of a transaction contains all entities of each entity class that is accessed. This means:

- We have to determine the entity classes accessed by a transaction.
- For all the entity classes involved in the transaction, we have to determine the cache region size. The cache region size of a particular entity is the size when all entities of the class (i.e. the entire table) are cached.

We calculate the space S_0 an entity class takes in memory when the entire table corresponding to the class is cached as below:

$$D_0 = Number\ of\ rows\ in\ the\ corresponding\ database\ table$$

$$E_0 = Size\ of\ one\ persistence\ entity\ instance\ in\ the\ cache$$

$$S_0 = Size\ of\ the\ cache\ region\ corresponding\ to\ the\ persistence\ entity\ class$$
$$= D_0 \times E_0$$

A point to note here is that the object might not be stored in the cache region in object form. For example, Hibernate, which is the default implementer of the JPA in application servers like JBoss, dehydrates the persistence entity instances before storing them in the second level cache.

The entity classes involved in a transaction can be determined by either analyzing the logs at the application server level or by parsing the application code. We take the later approach to determine the entity classes involved in each transaction type.

We can easily get the number of rows in the database table of a particular entity by using the SQL statement

```
Select count(*) from tableName;
```

Getting the size of a single entity in the second level cache for the particular cache we use is discussed in Chapter 4.

Once we have both the row count and the size of a persistence entity, we can easily get the cache region size of that entity class.

A transaction's working set size $WS(T_i)$ can then be calculated as the sum of the sizes of all persistence entities involved in it.

$$WS(T_i) = S_0 + S_1 + S_2 + \cdots + S_n$$

Where

$WS(T_i) =$ *the working set size of Transaction* T_i *and*

$S_0, S_1, S_2, \ldots, S_n$ *represent the respective size of the different persistence entity classes involved in transaction* T_i

3.1.3 Grouping requests

After knowing the size and content each transactions type accesses, we need to group the transaction types in such a way that

- We have an efficient utilization of the cache space available on all the application servers.
- We maximize the cache hit rate on each application server.

We also need to make sure that the transactions grouped, fit the cache space available on a single application server. This will benefit us as it will make sure that there are minimal

cache evictions. Having minimum cache evictions means that the user requests find the requested data more often in the cache. Application performance benefits by this because there are fewer calls to the database, less network traffic and less processing load on the database server.

To group transactions we use a content-aware bin-packing algorithm. Bin packing [23] has numerous applications and has been widely used in memory allocation algorithms at the operating system level. It provides us with a very good approach to group transactions in such a way that they fit in the cache space available at a single server.

We choose the well-known *best fit decreasing bin-packing* algorithm and modify it to make it content-aware. The bin packing algorithm used here takes as input the transactions along with their working sets and uses this information to group them. It also needs an estimate of the cache space available on an application server which it considers as the available bin capacity when performing bin-packing.

If several transaction types within a group access the same content (i.e., the same table), the algorithm accounts for this shared content. For example if we have two transactions T1 and T2 which access entity classes `Customers, Items, Regions` and `Customer, Bids, Items` respectively, then the combined working set size of the two transactions will be the sum of the size of the `Customers, Items, Regions` and `Bids` persistence entity classes. The shared content of `Customers, Items` is considered only once as both the transactions T1 and T2 would reference the same cache region of `Customers` and `Items`.

Transaction types are matched to a bin such that there is maximal overlap of contents within a bin and there is minimal space left after packing the transaction into the bin. Here, a bin refers to the second level cache on an application server.

There is a possibility that after performing bin-packing a transaction type whose size exceeds the cache capacity available on any server remains unpacked. Such transactions are overflow transactions and are assigned their own bin.

The pseudo code for bin-packing is described in Figure 3-1:

The bin packing algorithm groups transactions and maps them to a certain bin. Assume that the bin packing algorithm requires N_0 bins to pack all transactions (N_0 includes one bin for each oversized transaction). Then in order to do effective cache-aware load balancing we would need at least N_0 number of application servers. In a configuration where we have less than N_0 applications servers, multiple transaction groups need to be assigned to a single application server. This will cause cache contention and could result in poor performance and is not desirable. Also, for every oversized transaction we expect suboptimal cache hits.

Cache Aware Best Fit Decreasing Bin Packing Algorithm

Input:

$transaction[\,]$ `// array of transaction types`

Algorithm:

```
1:  sortDescending (transaction[ ])   // sort the transactions in the
    decreasing
                                          order of their size
2:  unAllocatedTransactions[ ] ← ∅    // initialize
3:  n ← transaction.length            // number of transaction types
4:  for i = 1 to n do
5:     if ( ∄ a bin with free space ≥ transaction[i].spaceNeeded *)
6:           unAllocatedTransactions.add(transaction[i])
7:           continue;
8:     end if
9:    maxIntersectionBins[ ] = getBinsWithMaxCommonContent(transaction[i])
10:   selectedBin =
            getBinWithMinLeftOverSpace(maxIntersectionBins[ ], transaction[i])
11:   selectedBin.pack(transaction[i])
12:   maxIntersectionBins ← ∅
13:   selectedBin ← ∅
14: end for
```

*For a particular bin,the space needed by a transaction is calculated as:
$(transaction.size - (transaction.content \cap bin.content).size)$

Figure 3-1 - Cache Aware Bin Packing Algorithm

3.1.4 Forwarding requests

After the bin packing process, the load balancer receives a mapping that indicates which transaction group is handled by which application server. When the client sends an `HTTP`

request, the load balancer figures out the transaction group to which this request belongs. It then looks up the mapping of transaction group to application server to forward the request to a particular node.

3.1.5 Replication

The content aware bin-packing algorithm described here makes the best possible effort to divide the transactions into groups which have disjoint working sets. In other words, the transactions belonging to different groups have minimal or no overlap in terms of common entities accessed. The degree of overlap depends on the type of transactions that exist in the system. It can very much be the case that there are no common entities accessed by different transaction groups. But there is also a possibility that entities exist, that are accessed by more than one transaction group. In the later case, the common entity classes will be cached by all the responsible application servers causing replication of those cached entries. In an environment where both read and update transactions exist, additional mechanisms will be needed to need to ensure the replicated cache content is consistent across the application servers. In this thesis, we are concentrating on read-only workloads, and do not consider consistency issues.

3.2 Dynamic Cluster

In the previous section we discussed a cache aware load balancing approach where we had a fixed number of applications servers in the cluster.

Furthermore, the proposed bin packing algorithm grouped transactions without any regard to how resource intensive the transaction type is. It could very much be the case that it groups very resource heavy transaction types together and maps them to the same application server. This might cause the application server to be overloaded compared to

the other servers in the cluster. In order to avoid this problem, the load balancer can assign some of the resource intensive transactions handled by this over-loaded application server to other application servers. Alternatively one can introduce new application servers into the cluster and have them share the load with the most loaded one.

The first approach will introduce cache contention leading to cache evictions. This reduces the effectiveness of the content-aware load balancing approach. Therefore, we do not further consider this solution. Instead we focus on the later approach, where new application servers are introduced as needed to share the load with the most loaded one. Adding node in this way leads to the formation of application server groups.

More precisely, an application server group is a sub-cluster of application servers responsible for handling the same transaction group. In this configuration, as the time progresses, we expect that all the nodes in an application server group will host copies of same objects namely the ones accessed by the transaction group. This means, we have cache content distributed across the application server groups as they are handling different transaction groups, and we have cache content replicated among the nodes in an application server group.

The approach discussed here means that the load balancer should be able to handle the addition or removal of new application servers on the fly. In our infrastructure, when the application servers start, they register themselves with the load balancer. After becoming aware of the new application server the load balancer starts dispatching requests to it. Also, in our configuration, the application servers broadcast information about the application lifecycle events to the load balancer. These events include application deployment/un-deployment, node shutdown etc. Such information enables the load balancer to do a transparent failover to another application server in case a node goes down, an application is in error state or the application has been un-deployed. In our implementation, we use the mod_cluster [24] load-balancer provided by JBoss. It provides already some features that are useful in our context. For example, it allows the

registration and deregistration of servers. Furthermore, it offers a load aware load-scheduling.

In a dynamic configuration, the load-balancer has to perform two further tasks. First, given a replication group with more than one application server, it has to decide to which application server to send the requests for the corresponding transaction group. In our system we take advantage of the load-aware functionality of mod_cluster.

Second, given the number of application servers and the number of transaction groups, the load-balancer has to decide how many application servers to assign to this group. This assignment should be dynamic depending on the current workload. We will discuss both features and their implementation in detail.

3.2.1 Load Distribution within an Application server Group

As there can now be several nodes in an application server group, the load-balancer has to decide to which node to send a particular request. For that, it uses a load-aware approach. This functionality is already provided by the mod_cluster [24] infrastructure. It calculates the load at the application servers and then sends it periodically to the load-balancer. In our implementation the load balancer then uses this information to determine where to send a request. The load balancer then sends the request to the least-loaded node in the application server group.

3.2.1.1 Load Calculation

The load calculation function at the application server can take into account multiple load metrics to calculate a cumulative load factor (`lbfactor`). For calculating the `lbfactor`, we configure it to take into account the CPU load of the application server

and the number of connector threads in the JBoss thread pool that are busy in servicing requests. The metrics are explained in detail later.

The load metrics can be assigned weights. Weights signify the importance of that metric while calculating the overall load factor. For example, a metric assigned a weight of 2 will influence the load factor twice as much as a load metric which has an assigned weight of 1.

As given in [25], the cumulative load factor is calculated by the following formula:

$$Load_{total} = \frac{L_1 * W_1 + L_2 * W_2 + \cdots + L_i * W_i + \cdots + L_N * W_N}{W_1 + W_2 + \cdots + W_i + \cdots + W_N}$$

Where,

L_i is the load for a particular metric and

W_i is the weight of a particular metric.

$Load_{total}$ is the cumulative load. Each application server sends periodically it's $Load_{total}$ as `lbfactor` value to the load-balancer. Mod_cluster uses a decay function to calculate the load value of each metric. The decay function takes into account the history of the load values of that particular metric. This approach reduces the effect of spikes experienced in the load values of a particular metric and provides a very accurate estimate of the load of that metric. It also minimizes the effect of noise in load measurement.

The load of a particular metric is calculated as below:

$$L_i = \left({L_{i0}}/{d^0} + {L_{i1}}/{d^1} + {L_{i2}}/{d^2} + {L_{i3}}/{d^3} + \cdots + {L_{ik}}/{d^k} + \cdots + {L_{iH}}/{d^H} \right) \times (1 + d + d^2 + \cdots + d^k + \cdots + d^H)$$

or more concisely as

$$L_i = \left(\sum_{k=0}^{H} {L_{ik}}/{d^k}\right) \times \left(\sum_{k=0}^{H} d^k\right)$$

Where

L_i is the value of a particular load metric, d is the decay factor and H represents the number of historical load factor values to take into account while calculating the load.

In the above equation, to if we do not want the historical values be taken into consideration while calculating the load metric value we can set $H = 0$. In our set up, we set $H = 10$ while calculating the load factor for a particular metric.

Also, in our implementation we have assigned the CPU load metric a weight of 2 and busy connector thread metric a weight of 1.

3.2.1.2 CPU Usage Load Metric

The CPU load is being calculated by using the function

ManagementFactory.getOperatingSystemMXBean().getSystemLoadAverage()

of Java. The above function is supported by Java 1.6 and above versions and returns the CPU usage of a particular machine.

3.2.1.3 Busy Connector Thread Load Metric

Application servers typically keep a thread pool on standard ports for accepting requests. This is done to ensure that many requests can be served simultaneously. The load balancer in our architecture forwards the requests it receives to the port at 8009 where the

application server keeps a connector thread pool. If all the threads in this thread pool are busy servicing requests then the client either experiences a large response time or the request could time out.

Therefore, the connector thread pool usage is an important metric to be considered while calculating the application server load. In our case, the connector thread pool load is the percentage of connector threads from the thread pool that are busy servicing requests.

The load balancing algorithm for dynamic cluster is shown in Figure 3-2.

3.2.2 Application Server Group Configuration

As discussed previously, application server groups consist of one or more application servers. For our dynamic cluster configuration, we need a metric which reflects the load of an application server group. This metric will help us decide to which group we should add the newly introduced node in the cluster. At this same time, this metric will also aid us while performing re-allocation of nodes to maintain equitable distribution of load among the groups.

3.2.2.1 Calculating and Comparing Group Loads

An application server group load is calculated by averaging the load balance factor of all the nodes present in the group. For example, if there is a group which has three nodes with load balance factors as 30, 45, 75 respectively, the load of the group will be 50.

When comparing the loads of application server groups, we compare their average loads. For example, to select the group with maximum load we return

$Max\{average_{load}(group_1), average_{load}(group_2), \cdots\cdots, average_{load}(group_n)\}$

as the most loaded group.

Load Balancing Algorithm for Dynamic Cluster

```
Input:
        request    // http request

Output:
        responsibleNode // application server responsible for handling the
                                        request

Algorithm:

 1:    transactionType = getTransactionType(request.queryString)
 2:    applicationServerGroup = getResponsibleASGroup(transactionType )
 3:    responsibleNode ← ∅
 4:    responsibleNode = getLeastLoadedNode(applicationServerGroup)

 5:    if responsibleNode == ∅
 6:        return leastLoadedNode from the entire cluster
 7:    end if

 8:    if responsibleNode is usable
 9:        return responsibleNode
10:    else
11:
        if (failover to other least loaded node in applicationServerGroup is possible)
12:           responsibleNode = other least loaded node in applicationServerGroup
13:           return responsibleNode
14:       else
15:           return leastLoadedNode from the entire cluster
16:       end if
17:    end if
```

Figure 3-2 - Load Balancing algorithm for a dynamic cluster

3.2.2.2 Addition of node

We assume that at system start-up there exist at least as many servers as there are transaction groups. That is, for each transaction group there exist an application server group with at least one application server. From there, whenever a new application server is added to the cluster, the load balancer adds the server to the group with the maximum load.

3.2.2.3 Re-allocation of Nodes among application server groups

In the above set up it can be the case as the system is running that an application server group is more loaded as compared to the other groups. In that case it would make sense to perform re-allocation of nodes so that there is a more even distribution of the load among the groups in the cluster and no group becomes the bottleneck. To perform this re-allocation we have a watchdog thread which keeps on monitoring the load of each server group and when the load of a certain group rises above a certain threshold ($Threshold_{Max}$) re-allocation of nodes from the least to the most loaded group happens.

Instead of using the current load statistics of an application server group to determine the least loaded group, we calculate the future load of the group after a node is removed from it and then select the one which will have the least load. For example: If we have 3 application server groups $group_1$,$group_2$, $group_3$. $Group_1$ has 3 machines with load 40, 50, 60 and average load of 50; $group_2$ has 2 machines with load 45, 45 and average load of 45 and $group_3$ has load of 80, 85, 90 and average load of 85. In this case if a node is removed from $group_1$ then the future load it will experience will be (40+50+60)/2 = 75 and if the node is removed from second group then it will have load of 45+45/1 = 90. Hence it makes more sense to take a node from $group_1$ and assign it to $group_3$. This will ensure that re-assignment will not make the donor group the bottleneck in the system. Furthermore, if for all the groups the future load is greater

than $Threshold_{Max}$, then no application server is taken out. This ensures that the donating group does not become the bottleneck if node were removed from it.

Finally, if any application server group is loaded below a certain threshold value $Threshold_{Min}$, the watchdog thread tries to perform re-allocation between this group and the most loaded application server group.

The algorithm for the watch dog thread is shown in Figure 3-3:

Watch Dog Thread Algorithm

```
Algorithm:

 1:   applicationServerGroup[ ] = getServerGroupsWithLoadAbove(Threshold_max)
 2:   n ← applicationServerGroup.length

 3:   for i = 1 to n do
 4:     flag ← false
 5:     while (applicationServerGroup[i].load > Threshold_Max)
 6:       donorGroup = getDonorGroup( )  // returns a group with load less
                                            than Threshold_Max after removing
                                            least loaded application server
                                            from it
 7:       if (donorGroup == null)
 8:         flag ← true
 9:         break                          // no node can donate node
10:       else
11:         node ← getNodeFromGroup(donorGroup)  // returns least loaded
                                                    Node from donorGroup

12:         move node from donorGroup to applicationServerGroup[ i ]

13:         updateLBstatus(donorGroup)          //  updates lbstatus factor of  the
                                                    donorGroup by distributing the
                                                    lbfactor of donated node
14:       end if
15:     end while

16:     if (flag == true)
17:       break                      // since no donor group available therefore exit
18:     end if

19:   end for
```

Figure 3-3 - Watchdog Thread Algorithm

Chapter 4

Performance Results and Evaluation

In this chapter we present the evaluation of our CALB algorithm. Here, we first describe the experimental test bed and its specification. Then, we describe the benchmark used to evaluate our load balancing algorithm in a dynamic content environment. In the evaluation section, we discuss the methodology used to evaluate the system and compare our implementation with other load balancing algorithms. Later, we present the various figures and statistics obtained and discuss the significance of those numbers.

4.1 Experiment Test-bed

4.1.1 Cluster Specification

The machines and the LAN network in the cluster have the following specification:

CPU	Intel(R) Pentium(R) D CPU 2.80GHz With 2 MB L2 Cache.
RAM	1 GB DDR
Hard Disk	80 GB, 7200 RPM
Network Card	1000 Mb/s
Switch	100 Mb/s

Table 1 - System Specification for Experiments

4.1.2 System Architecture

The experiment test-bed is set up as a multi-tier system. We have a client tier which communicates with a load balancer. Behind the load balancer we have a cluster of application servers and a database server. The back-end cluster is set up on a local LAN.

In our set up, every machine except the load balancer used in the experimental evaluation has *Ubuntu* [26] *9.04* with kernel version *2.6.28-19-generic* installed. We are using JBoss [8] version *6.0.0.M2,* a Java EE compliant application server installed on the middle tier. In JBoss, we use Hibernate as the ORM mapping facility. For persistence, we use PostgreSQL [27] version *8.4.1* as our database server.

The load balancer set up in our case is a dispatcher level load balancer. We use Apache httpd [28] web server integrated with the mod_cluster [24] module from JBoss as our load balancer. The operating system installed on the load balancer machine is Windows Server 2008 [29]. The apache httpd and mod_cluster module are compiled and installed

for the said platform. In the set up, the load balancer is running as a reverse proxy where it acts as the gateway for the servers in the cluster. It is the sole point of contact for all the client machines and fetches the content from the back end nodes transparently. We have made changes in the mod_cluster module for installing our load balancing algorithms.

In our configuration, the load balancer communicates with the back end application servers through the AJP protocol. AJP stands for Apache JServ Protocol. It is a binary protocol that can forward inbound requests from a web server to application servers which sit behind the web server [30]. AJP is frequently used in load-balanced set ups where one or more front-end web servers forward requests to multiple application servers in the back end. Communication between application server and the Apache httpd web server (load balancer in our configuration) is more optimal in terms of bandwidth when using AJP protocol compared to the http protocol.

In our middle tier which is hosting the Java EE based application server (JBoss) for our configuration, we use Hibernate version 3.3.0 as the ORM framework. EHcache [31] version 2.0.1 is used as our 2nd level cache provider.

For our implementation we mainly change mod_proxy_cluster module of mod_cluster from JBoss. Some changes are also made to mod_proxy module of Apache httpd. We also change the EHcache source code for our CALB algorithm. Except for all these components which we enhanced for our load-balancing strategy, all other components were used off-the-shelf without any modification.

In all the machines, we have Java from Sun (now Oracle) with version 1.6.0_16 installed.

The system architecture of our experiment test bed is shown in Figure 4-1.

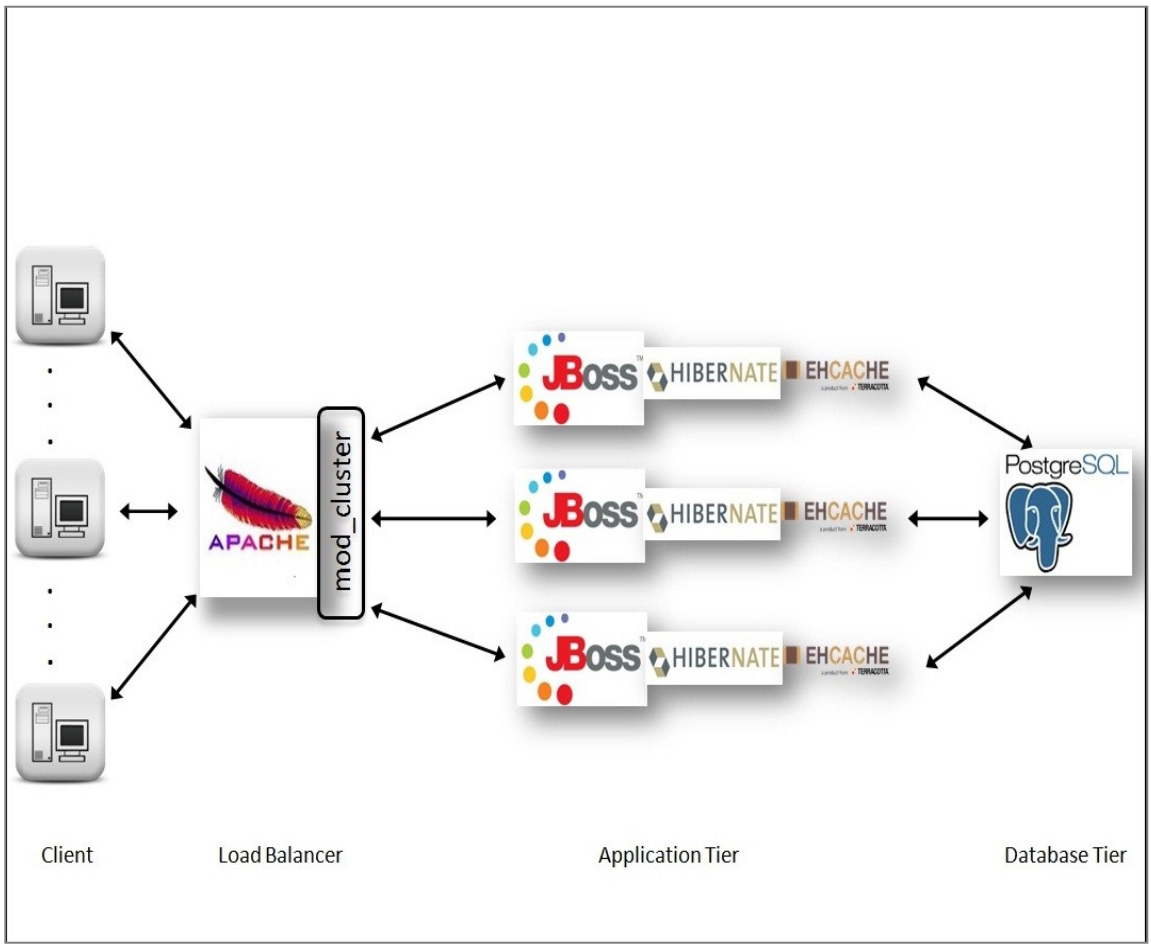

Figure 4-1 - Test-bed Architecture

4.1.3 Benchmark

We use the RUBiS [32] benchmark to evaluate our implementation. RUBiS is described as below on its homepage [32].

"RUBiS is an auction site prototype modeled after eBay.com that is used to evaluate application design patterns and application server's performance scalability."

RUBiS Benchmark description

RUBiS is a popular benchmark widely used to evaluate the performance of web and enterprise applications and servers [33].

RUBiS models and implements the basic functionality of an auction site, i.e. selling of items, browsing and bidding on items. As described in [33], RUBiS provides three types of user sessions:

- Buyer Session: In this type of interaction, the user needs to log into the system with a username and password before it can carry out any transaction. The user needs to register itself for the first time before it is allowed to log in and use the system. In this session, a user typically puts bids on items and buys them.
- Visitor Session: This type of user interaction needs no log-in into the system but the services/transactions allowed to this user are limited. For example: the user cannot buy or bid on items while in a visitor session.
- Seller Session: For a seller session, like for a buyer session, the user needs to log in into the system. The typical transactions carried out in this session are reserving a price for an item, etc.

In RUBiS, a single session consists of multiple interactions of a single client with the server. The RUBiS client emulates a client/customer of the auction website by spawning a thread which then opens a persistent `HTTP` connection to the server. The connection is cleaned up at the end of the session. After having received a response from a previous session and before initiating the next request, the spawned thread waits a certain time, which is set by the think time property in RUBiS. As stated on RUBiS homepage [32]

"The think time and session time is generated from a negative exponential distribution with a mean of 7 seconds and 15 minutes, respectively."

There is also a possibility to set the think time from a configuration file.

The relational database schema of RUBiS consists of 7 tables namely: *users, items, categories, regions, bids, buy_now* and *comments*. Also, the RUBiS benchmark defines 26 different request types. The type of the next request that the client will submit to the server is determined by a transition matrix fed to RUBiS through a configuration file. The transition matrix specifies the probability of going from one state to the next state. Here, state means a request type. In other words, the request execution pattern in RUBiS can be thought of as a directed graph where different vertices represent the different states/request types and the edges connecting the vertices as possible transitions from a particular vertex. The weight on the edges can be thought of as the probability of going from one state to the next.

The RUBiS benchmark defines several types of workload mixes. For instance, the browsing mix is made of only read-only transactions while the bidding mix has 15% read-write transactions. For the purposes of our evaluation, we only use the browsing (read-only) mix. Our main target is to evaluate the benefit of a cache aware load balancing algorithm on caching at the application server level. Thus, a read only mix gives us the best case scenario for evaluating it. In a read-only workload, we are likely to have higher cache hit rate compared to a read write mix where an update has to go to the database to update the record. Hence a write request cannot take advantage of the cached data. At the same time, having a read-only mix simplifies the system design as one does not have to consider the inconsistencies that can occur if several servers cache the same object.

The client emulator in RUBiS is written in Java. In RUBiS, the client execution profile is defined by a properties file. One can vary the workload by varying the number of clients. The client emulator in RUBiS generates various useful statistics like average CPU load, average network I/O, etc, and produces graphs for CPU utilization, memory use, network interface traffic, disk, I/O, etc over time. RUBiS collects all this information from the machines participating in the experiment by using `ssh` and calling the *sar* linux/unix utility. It then uses the *ksar* [34] utility and various other scripts to generate the graphs.

In addition to all this, for each transaction type, RUBiS also displays statistics like average, minimum and maximum execution response time.

To allow RUBiS to `ssh` into the cluster machines and monitor the nodes during the experiment, we use PortTunnel software [35]. The software provides the service of port forwarding on the Windows Server 2008. We need port forwarding here because our entire cluster is on a local private LAN which is abstracted by a load balancer having a public IP. The port forwarding service then forwards the TCP/IP packets that are send to a configured port on the load balancer to the desired machine and port of our cluster machine.

The RUBiS application, which gets deployed in an application or web server, has two content generating components. One is the static part which contains static `HTML` files and images. The other is the dynamic content generator which involves interaction with the database. For handling dynamic content RUBiS provides three different implementations. The three implementations use the PHP Web-scripting language, Java Servlets and Enterprise Java Beans respectively. In our experiments, we use the Servlet implementation of RUBiS. The Servlet implementation is packaged as a WAR file and deployed in the JBoss application server.

4.2 Evaluation and Results

In this section we talk about the evaluation methodology adopted and then discuss the results of the different test runs. In regard to evaluation methodology, we discuss the way the system was configured for the test runs.

In the evaluation section we compare our implementation of the CALB with the well known content aware load balancing algorithm LARD [18] and with round robin scheduling. We later present the results and discuss the observations made.

4.2.1 Methodology

For our evaluation, as mentioned before, we only consider the read-only workload mix of RUBiS. The read-only workload mix of RUBiS consists of the following 10 transaction/request types:

1. BrowseItemsInRegion
2. BrowseCategories
3. BrowseItemsInCategory
4. BrowseCategoriesInRegion
5. BrowseRegions
6. SelectCategoryToSellItem
7. ViewItem,
8. ViewUserInformation
9. ViewBidHistory
10. AboutMe

The above transactions/request types have dynamic content needs, which involve accessing the database and fetching records matching the user query parameters. There are some request types in RUBiS which access static content like images and `HTML` files. A few examples of such requests types are: *Home, Register, Browse,* etc. Our cache aware load balancer does not focus on such requests. Such transaction types are forwarded to application servers based on the current load of the nodes (least loaded one). These requests could be included in our cache aware load balancing algorithm but the static files in our case are only a few Kilobytes in size. Therefore, we can safely ignore such requests in our consideration.

In our test configuration, the RUBiS database size is 60 Mb. Since the bin packing algorithm used in the load balancing algorithm targets the cache size available on an

application server, we need an estimate of how much space a row needs in object oriented format in the cache and then get an estimate of the size of the entire table in the second level cache. In the succeeding paragraph we describe how we estimate the table size.

We use EHcache as our second level cache provider. EHcache caches each table in its corresponding cache region, which means all tables in the database are cached in separate cache regions. This configuration is fed to EHcache through its ehcache.xml configuration file. To estimate the size of a cache element in memory we use API's of EHcache. EHcache has an interface method that provides an estimate of the size of an element in the cache region. We then calculate the table (cache region) size by multiplying the size of a single cache element with the number of rows in the database for the table corresponding to that cache region.

Also, for our purposes we need to have a way of restricting the entire cache size in memory. In other words, we should be able to dedicate a pre-fixed amount of memory in Kilobytes or Megabytes for the second level cache. Since we are running JBoss which executes in a Java Virtual Machine, the amount of memory that will be allocated for the second level cache only happens on demand and we cannot predefine a size. If the cache size exceeds the Java Virtual Machine heap size, we will get an out of heap error. Also, the memory usage varies in JVM as the ORM framework (Hibernate) has a first level cache which gets de-allocated after each session. Such behaviour makes the memory usage of other running instances in the JVM unpredictable. This means, we cannot control the cache size by varying the JVM heap parameters.

To control the amount of memory that EHcache uses for all of its cache regions we modify the EHcache source code. EHcache controls the cache region size in terms of the number of elements that can be accommodated in it. There is no way to specify the cumulative size in Kilobytes or Megabytes that EHcache should occupy in memory when it is running. An example configuration of a typical cache region in EHcache is:

```
<cache name="edu.rice.rubis.hibernate.User"
    maxElementsInMemory="2000000"
    eternal="true"
    timeToLiveSeconds="120000"
    overflowToDisk="false"
    memoryStoreEvictionPolicy="LRU">
</cache>
```

Here, the *maxElementsInMemory* attribute of the *cache* element controls the maximum number of allowed elements in the cache region named "edu.rice.rubis.hibernate.User".

Since EHcache 2.0, certain aspects of the cache configuration can be changed dynamically at runtime [36], namely

- *timeToLive*
- *timeToIdle*
- *maxElementsInMemory*
- *maxElementsOnDisk*
- *memory store eviciton policy*

We take advantage of this feature of EHcache 2.0 to set a cap on the memory in Kilobytes used by EHcache. To this end we implemented a *cache listener* which gets notified on the following events:

- Notify Element Removed
- Notify Element Put
- Notify Element Updated
- Notify Element Expired
- Notify Element Evicted
- Notify Remove All

To set the cap on the memory size of EHcache, we use the "Notify Element Put" event. Inside the event, on an *element put* to any cache region, we check the memory used by

multiplying the number of elements present in all the cache regions by their respective element size. We dynamically set the element count as soon as the cache limit specified in Kilobytes is reached. At this juncture, we can say that the cache has been filled and any cache put will lead to a cache eviction

The cache size in our case is 33 Megabytes for each application server. The in-memory size of the database is around 100 Megabytes.

4.2.2 Results

The cache aware bin packing logic gives the following grouping of transactions. Each transaction group is then matched to an application server group. This means in our evaluation we have 3 application server groups each responsible for one of the below transaction groups.

Group 1	**BrowseItemsInRegion** **BrowseItemsInCategory** **BrowseCategoriesInRegion** **BrowseRegions** **BrowseCategories**
Group 2	**ViewUserInformation** **SelectCategoryToSellItem**
Group 3	**AboutMe** **ViewItem** **ViewBidHistory**

In the above transaction groups we have some degree of replication of content between the groups as the groups formed by the bin packing are not completely disjoint. For

example: Group 1 and Group 2 both access the entity class `Categories` is common. This means `Categories` entities can be replicated between the application servers responsible for Group 1 and Group 2. Between Group 2 and Group 3 entity class `Comments` and `Users` are common. There is no persistence entity class which is replicated across all the transaction groups.

We evaluate our CALB against LARD and Round Robin. Recall that LARD forwards same requests to the same application server without any regard to how much data that server can cache. On the other hand, round robin forwards requests one by one to the application server cluster nodes.

Here, we perform different test runs by gradually increasing the number of clients in the RUBiS client emulator. In our evaluation, the system reaches saturation at around 1300 clients. It is at this point we observe maximum system throughput. Increasing clients beyond this number results in a worse system response time. We present detailed results for this test run and present a summarized comparison for different parameters (throughput and response time) for different number of clients in the summary section of this chapter. For 1300 clients, we note the response time of individual transactions and make comparisons with the ones obtained for different load balancing algorithms. The duration of each test run is 3600 seconds. The warm up phase in our experiments had duration of 120 seconds.

Tables 2, 3, 4 show the response times of individual transactions and also shows the overall response time for three load-balancing algorithms. Tables 2, 3, and 4 also show the throughput obtained for the above mentioned load balancing algorithms.

Request Type	Minimum Response Time	Maximum Response Time	Average Time
BrowseCategories	9 ms	12458 ms	202 ms
BrowseItemsInCategory	9 ms	56054 ms	1292 ms
BrowseRegions	11 ms	12166 ms	201 ms
BrowseCategoriesInRegion	10 ms	10023 ms	178 ms
BrowseItemsInRegion	8 ms	48469 ms	424 ms
ViewItem	9 ms	93571 ms	327 ms
ViewUserInformation	10 ms	45380 ms	221 ms
ViewBidHistory	9 ms	21185 ms	240 ms
SelectCategoryToSellItem	0 ms	0 ms	0 ms
AboutMe	0 ms	0 ms	0 ms
Average throughput	**167 req/s**		
Completed sessions	**1117**		
Average Response Time	**748 ms**		

Table 2 - CALB - 1300 clients

Request Type	Minimum Response Time	Maximum Response Time	Average Time
BrowseCategories	9 ms	48117 ms	298 ms
BrowseItemsInCategory	9 ms	51694 ms	1540 ms
BrowseRegions	11 ms	33054 ms	298 ms
BrowseCategoriesInRegion	9 ms	33128 ms	277 ms
BrowseItemsInRegion	7 ms	46716 ms	576 ms
ViewItem	10 ms	66655 ms	403 ms
ViewUserInformation	10 ms	45046 ms	316 ms
ViewBidHistory	9 ms	33828 ms	314 ms
SelectCategoryToSellItem	0 ms	0 ms	0 ms
AboutMe	0 ms	0 ms	0 ms
Average throughput	**157 req/s**		
Completed sessions	**1066**		
Average Response Time	**907 ms**		

Table 3 - LARD - 1300 clients

Request Type	Minimum Response Time	Maximum Response Time	Average Time
BrowseCategories	7 ms	36122 ms	338 ms
BrowseItemsInCategory	7 ms	93849 ms	1771 ms
BrowseRegions	8 ms	39790 ms	328 ms
BrowseCategoriesInRegion	7 ms	45046 ms	353 ms
BrowseItemsInRegion	6 ms	45198 ms	646 ms
ViewItem	9 ms	90295 ms	483 ms
ViewUserInformation	8 ms	55582 ms	376 ms
ViewBidHistory	7 ms	47111 ms	364 ms
SelectCategoryToSellItem	0 ms	0 ms	0 ms
AboutMe	0 ms	0 ms	0 ms
Average throughput	**152 req/s**		
Completed sessions	**1043**		
Average Response Time	**1050 ms**		

Table 4 - Round Robin – 1300 Clients

The below graph (Figure 4-2) presents the average response times of individual transactions in the three different load balancing algorithms. As is clearly evident, all the transactions have the least response time using the CALB algorithm, followed by LARD and then Round Robin.

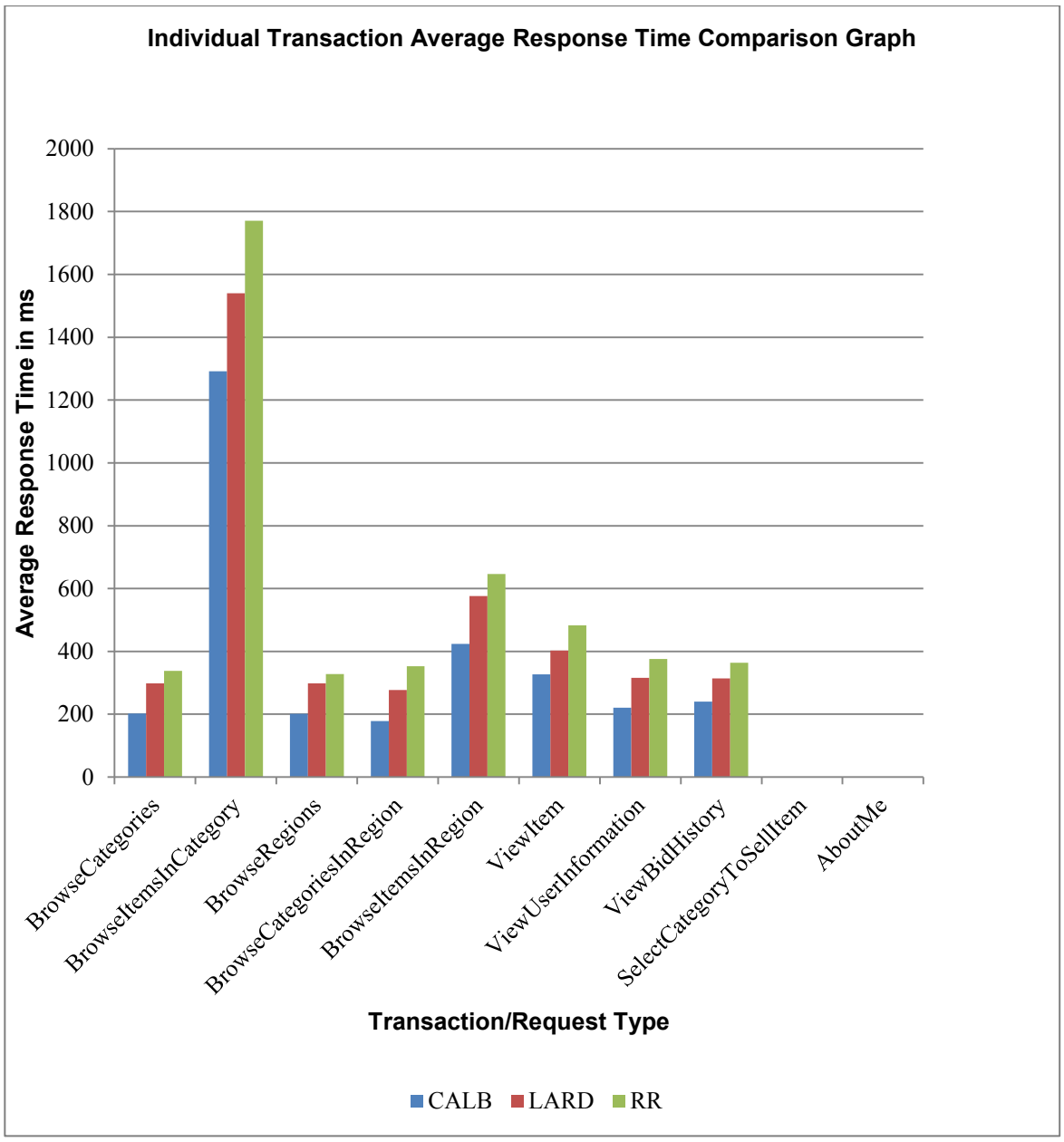

Figure 4-2 - Transaction Response Times

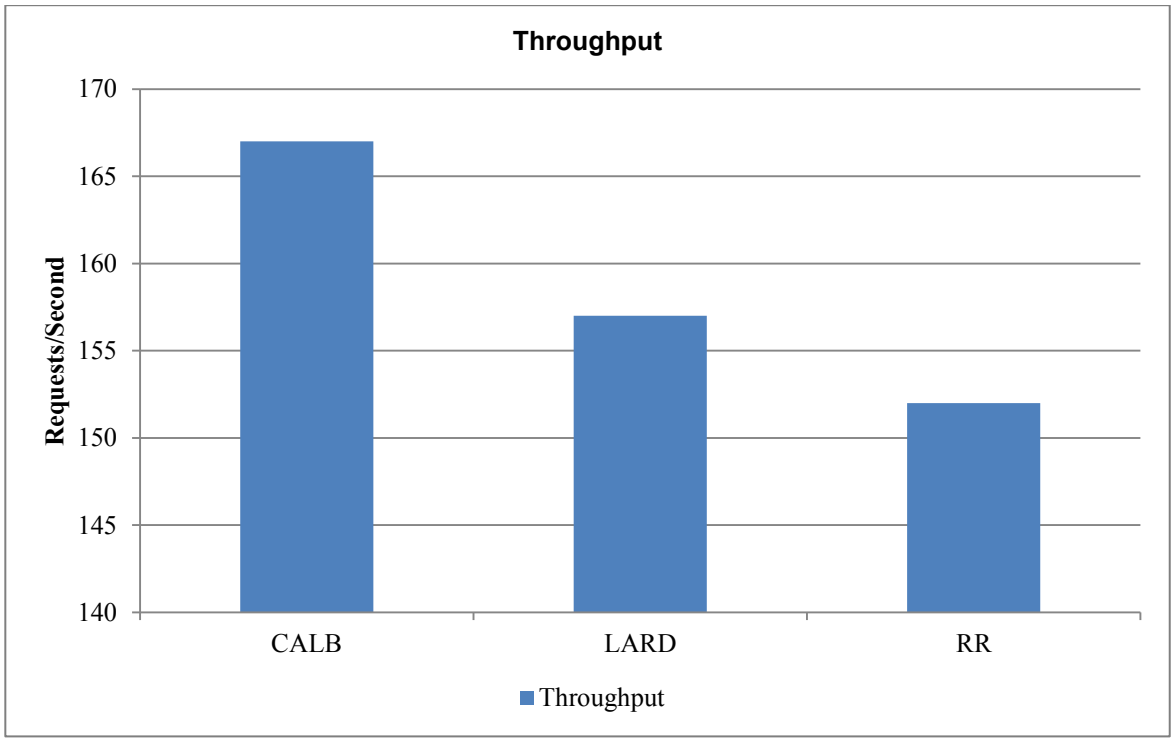

Figure 4-3 - Throughput

As shown in the above graph (Figure 4-3), CALB gives an improvement of 6.3 % in throughput over LARD and an improvement of 9.8 % over Round Robin scheduling algorithm.

For response time (Figure 4-4), CALB gives an improvement of about 17% compared to LARD and an improvement of 28% percent as compared to Round Robin. The decrease in response time can be attributed to the increase in the cache hit rate and fewer accesses to the database, which considerably reduce the response time for the client. Later, we present statistics which substantiate our reasoning.

Also, CALB completes more sessions than LARD and Round Robin. 51 more sessions are completed in CALB algorithm as compared to LARD, whereas 74 more sessions are completed in CALB compared to Round Robin.

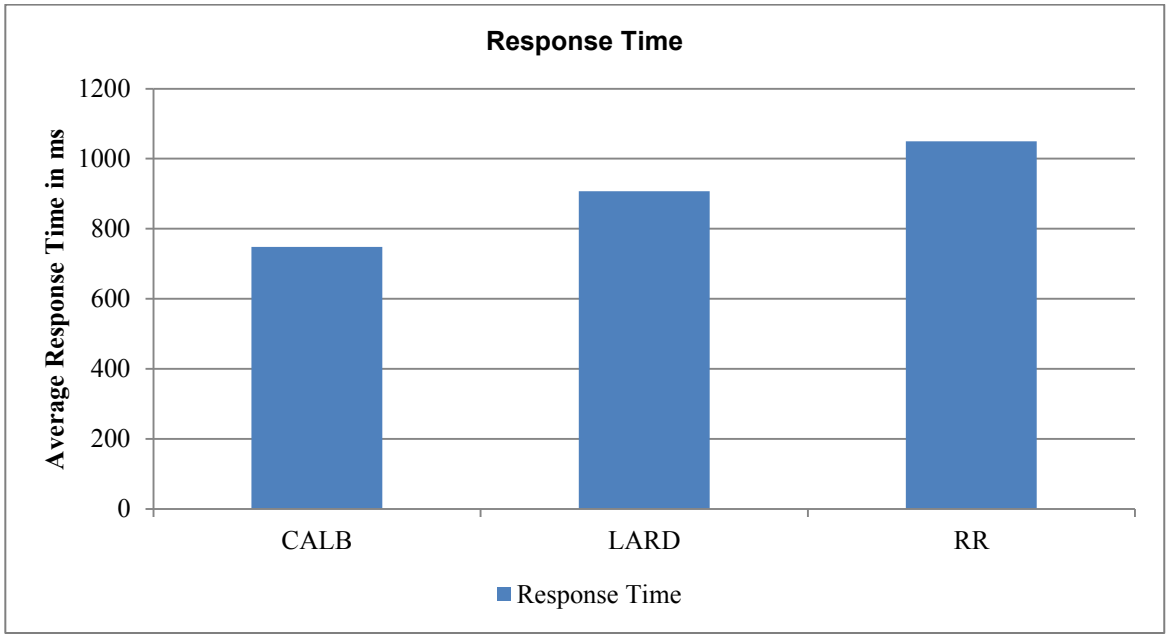

Figure 4-4 - Average Response Time

We also note the average network I/O at the database server. The four parameters reported by the sar utility for the network interface traffic in the three algorithms are given below (Table 5):

	CALB	LARD	RR
rxkB/s	15.62	24.34	27.75
txkB/s	39.61	80.84	94.52
rxpck/s	42.49	95.19	108.22
txpck/s	33.8	89.95	101.07

Table 5 - Average Network I/O for CALB, LARB and Round Robin at DB Server

Here

rxpck/s is the total number of packets received per second.

txpck/s is the total number of packets transmitted per second.

rxkB /s is the Total number of kilobytes received per second.

txkB /s is the Total number of kilobytes transmitted per second.

Our main figure of interest here is *rxKB*/s which gives the amount of data in kilobytes received by the database server per second. This data is mostly database query packets sent by the application servers to the database server. It is evident from the graph that CALB decreases the amount of database accesses by a great extent by making an effective use of cache.

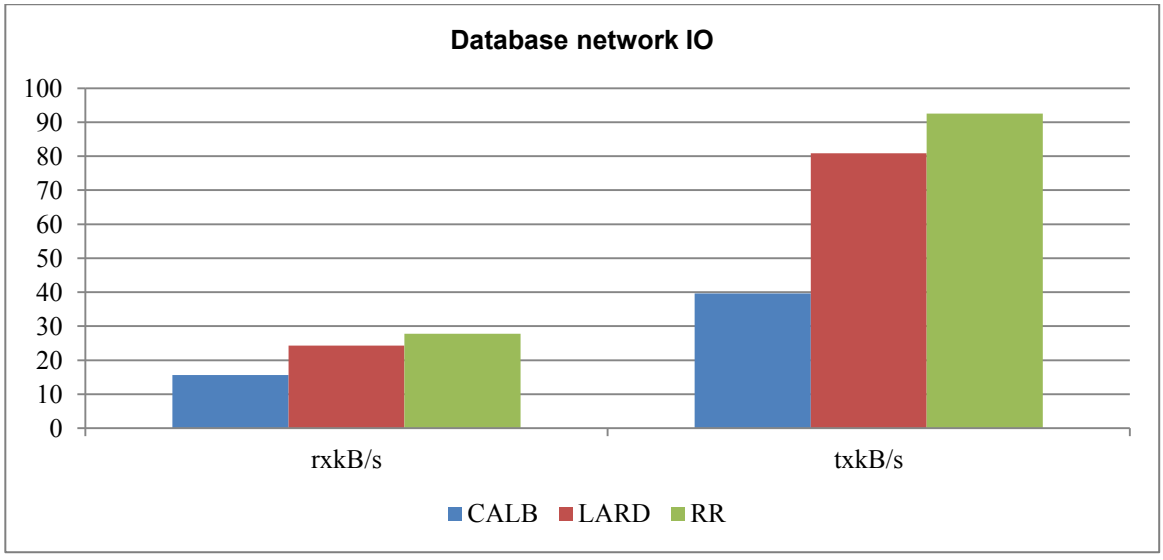

Figure 4-5 - DB network I/O

As we can clearly see, CALB results in a significant decrease in the traffic directed to the database (35% decrease compared to LARD). This indicates that a majority of requests in CALB are getting satisfied from the cache itself resulting in fewer accesses to the database and hence lower network interface traffic at the database. The huge jump in network traffic in case of LARD and Round Robin can be related to the cache evictions that occur in LARD and Round Robin. This happens because both LARD and Round Robin do not consider the working set size of the transactions in the context of the cache space at an application server while dispatching the transactions to them. This can result in multiple transactions be dispatched to an application server such that the entire working set of those transactions cannot be accommodated in the cache space available. This results in cache contention among the transactions dispatched to the same application server node, resulting in cache evictions later on. In Round Robin, this contention is even worse as it also completely ignores the type of data accessed by a transaction and its working set size. Thus, it performs the worst among the three load balancing algorithms in discussion here. Whereas, in CALB the working set of the transactions is accommodated in the cache space available on an application server. This results in zero cache evictions. On the other hand LARD does not take into account this criterion while dispatching requests. In LARD, eventually the cache will be overfilled resulting in cache evictions. This causes it to perform worse than CALB. Compared to Round Robin, LARD achieves a higher cache hit rate as it does take into account the type of data needed by a request while dispatching them to application server cluster nodes. This explains the better performance of LARD as compared to Round Robin. Round Robin on the other hand, is completely cache agnostic and blindly forwards the requests to the application server nodes. Because of this, Round Robin does not make an effective usage of the cache space available resulting in huge cache evictions and misses. This results in bad performance and explains why Round Robin performs worse than LARD or CALB.

To substantiate our reasoning, we also measure the number of cache evictions in the above test runs. The number of cache evictions in LARD is around 80,000 whereas the

number of cache evictions in Round Robin is around 100,000. In CALB we don't observe any cache evictions as the working sets fit the cache space available.

4.3 Summary

The above discussion clearly indicates that the CALB results in a significant decrease in the transaction response time compared to LARD and Round Robin. CALB not only increases the throughput but also reduces the network traffic that the database experiences because of cache misses. This reduces the processing load on the database, and thereby results in an overall scalable system at the database level. Such a load balancing algorithm allows the system to use the cache effectively and efficiently.

In Figures 4-6 and 4-7, we compare the response time and throughput for the different load balancing algorithms for 750, 1000, 1250, 1300 and 1500 clients.

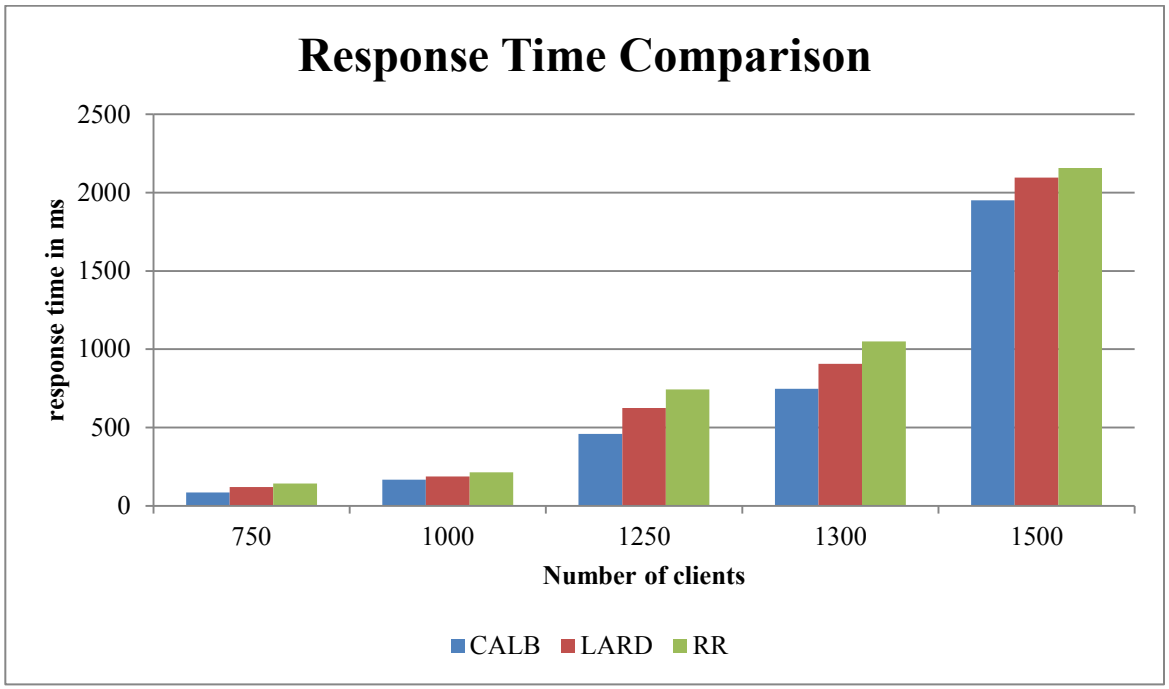

Figure 4-6 – Comparison of response time with varying number of clients

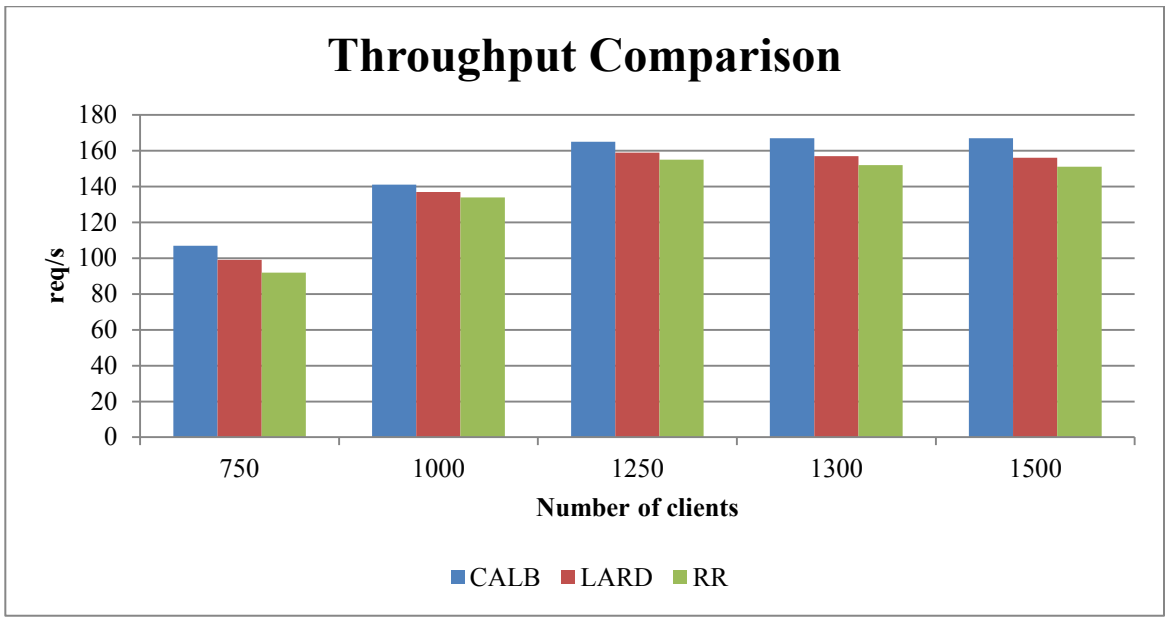

Figure 4-7 - Comparison of throughput with varying number of clients

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Scalable multi-tier solutions are crucial for database driven applications. As the businesses become more and more dependent on the internet to drive their commercial activities, a cost effective, scalable and highly available solution is needed. Different solutions have been presented in the literature to enhance the performance of the e-commerce and enterprise applications. A caching layer hosted by mirrored application servers, all of which share a single database, is a commonly deployed configuration to this end. In this thesis, we presented a cache aware load balancing solution (CALB) at the application server level, with the end goal of utilizing the cache space available on the application server cluster effectively. Our solution is integrated into a JEE server cluster. By identifying the different transactions in the applications and grouping them, the requests are divided into groups which have nearly disjoint content needs. This means less duplication of cache content over the application servers and much more effective usage of the caching component. We also extend the design of our cache aware load balancing solution to a configuration where the application server can start up and shut down while the system is running.

We have run experiments on our implementation and compared it with a content blind (round robin) and a content aware (LARD) strategy to demonstrate the effectiveness of our cache aware load balancer. In our evaluation, we observed a decrease of 17% in response time as compared to the LARD and a decrease of 28% in response time as compared to the round robin scheduling. In throughput, CALB resulted in an improvement of 6.3% compared to LARD and 9.8% compared to round robin. At the same time, the amount of network traffic in bytes received by the database server compared to LARD is decreased by 35%. This demonstrates that a larger proportion of requests are being satisfied from the cache and hence lesser network traffic is directed to the database server.

5.2 Future Work

A few possible directions for future work are discussed below:

The work presented in this thesis focused on a read-only workload for the purpose of evaluation. An evaluation against an update intensive workload would mean incorporating some consistency mechanisms at the cache level. Existing consistency mechanisms use a time-to-live approach, invalidation or polling. In [37], Cao *et al.* do a comparative analysis of the mentioned approaches on the performance of an application. A similar study can also be carried out on our infrastructure.

At the same time, in CALB, replication of content across caches is very minimal. It would be nice to take this into consideration while designing consistency mechanisms. Taking this feature of CALB into account would mean lesser amount of network traffic accounted by the invalidation messages sent across the network for maintaining cache content consistency.

Another future direction of work would be to automate the request type detection in the load balancer. Right now, the load balancer needs the client to identify the type of

requests it is handling. This can possibly be automated with some machine learning techniques or some sort of statistical analysis of the logs.

Bibliography

[1] Internet World Usage Stats. [Online] June 30, 2010. http://www.internetworldstats.com/stats.htm.

[2] Who We Are eBay Inc. [Online] http://www.ebayinc.com/who.

[3] Elnikety, S., S. Dropsho, and W. Zwaenepoel, *Tashkent+: memory-aware load balancing and update filtering in replicated databases*, in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 2007, ACM: Lisbon, Portugal. p. 399-412.

[4] The linux Virtual Server Project. [Online] http://www.linuxvirtualserver.org/.

[5] The Apache HTTP server project. [Online] http://httpd.apache.org/docs/2.1/mod/mod_proxy_balancer.html.

[6] The Java EE 6Tutorial. [Online] Oracle, October 2010. http://download.oracle.com/javaee/6/tutorial/doc/.

[7] The J2EE 1.4 tutorial. *The J2EE 1.4 tutorial.* [Online] http://download.oracle.com/javaee/1.4/tutorial/doc/.

[8] JBoss AS - JBoss Community. [Online] http://www.jboss.org/jbossas/.

[9] JOnAS Open Source Java EE Application Server. [Online] http://wiki.jonas.ow2.org/xwiki/bin/view/Main/.

[10] Glassfish Open Source Application Server. [Online] https://glassfish.dev.java.net/.

[11] Enterprise JavaBeans Technology. [Online] http://www.oracle.com/technetwork/java/index-jsp-140203.html.

[12] Java Persistence API - A Simpler programming model for entity persistence. [Online] http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html.

[13] Scalability. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Scalability.

[14] Cardellini, V., M. Colajanni, and P.S. Yu, *Dynamic Load Balancing on Web-Server Systems.* IEEE Internet Computing, 1999. **3**(3): p. 28-39.

[15] Ehcache. [Online] http://ehcache.org/EhcacheUserGuide.html.

[16] Luo, Q., et al., *Middle-tier database caching for e-business*, in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 2002, ACM: Madison, Wisconsin. p. 600-611.

[17] Gilly, Katja, Juiz, Carlos and Puigjaner, Ramon.**,** *An up-to-date survey in web load balancing, World Wide Web,* 2010, pp. 1-27.

[18] Pai, V.S., et al., *Locality-aware request distribution in cluster-based network servers*. SIGOPS Oper. Syst. Rev., 1998. **32**(5): p. 205-216.

[19] Cherkasova, L., *FLEX: Load Balancing and Management Strategy for Scalable Web Hosting Service*, in *Proceedings of the Fifth IEEE Symposium on Computers and Communications (ISCC 2000)*. 2000, IEEE Computer Society. p. 8.

[20] Zhang, X., et al., *HACC: an architecture for cluster-based web servers*, in *Proceedings of the 3rd conference on USENIX Windows NT Symposium - Volume 3*. 1999, USENIX Association: Seattle, Washington. p. 16-16.

[21] Cherkasova, L. and M. Karlsson, *Scalable Web Server Cluster Design with Workload-Aware Request Distribution Strategy WARD*, in *Proceedings of the Third International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS '01)*. 2001, IEEE Computer Society. p. 212.

[22] Iyengar, A., E. MacNair, and T. Nguyen. *An analysis of Web server performance*. in *Global Telecommunications Conference, 1997. GLOBECOM '97., IEEE*. 1997.

[23] Lodi, Andrea.**,** *Algorithms for Two-Dimensional Bin Packing and Assignment Problems, Universita degli studi di bologna. Bologna*, 1999. PhD thesis.

[24] Jboss mod_cluster. [Online] http://jboss.org/mod_cluster.

[25] mod_cluster Load Metrics. [Online] http://docs.jboss.org/mod_cluster/1.0.0/html/javaload.html.

[26] Ubuntu. [Online] http://www.ubuntu.com/.

[27] PostgreSQL. [Online] http://www.postgresql.org/.

[28] Apache Httpd. [Online] http://httpd.apache.org/.

[29] Windows Server 2008. [Online]
http://www.microsoft.com/windowsserver2008/en/us/default.aspx.

[30] AJP Wikipedia. [Online] http://en.wikipedia.org/wiki/Apache_JServ_Protocol.

[31] Ehcache - Performance at any scale. [Online] http://ehcache.org/.

[32] RUBiS. [Online] OW2 Consortium. http://rubis.ow2.org/.

[33] Amza, C., et al. *Specification and implementation of dynamic Web site benchmarks*. in *Workload Characterization, 2002. WWC-5. 2002 IEEE International Workshop on*. 2002.

[34] KSar. [Online] http://sourceforge.net/projects/ksar/.

[35] PortTunnel. [Online] http://www.steelbytes.com/?mid=18.

[36] Ehcache - Cache Configuration. [Online]
http://ehcache.org/documentation/configuration.html.

[37] Cao, P. and C. Liu, *Maintaining Strong Cache Consistency in the World Wide Web.* IEEE Transactions in Computing., 1998. **47**(4): p. 445-457.

[38] JPA Caching. [Online] http://weblogs.java.net/blog/archive/2009/08/21/jpa-caching.