Scalable object-based load balancing in multi-tier architectures

Sanket Manjul Joshipura

Master of Science

School of Computer Science

McGill University

Montréal, Québec, Canada.

August 2011

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

Acknowledgements

I am extremely grateful to my supervisor Prof. Bettina Kemme, for giving me the opportunity to work on this thesis. It would not have been possible to complete this thesis without her ideas, guidance and expertise. I am thankful to her for her support, encouragement and help throughout the two years that I have spent at McGill.

I am thankful to Rizwan Maredia, with whom I have collaborated on certain aspects whilst working on this thesis. It has been a pleasure to work with him. I would also like to thank Kamal Zellag, Neeraj Tickoo and Shamir Ali for sharing their experiences and knowledge at various stages during this thesis. I also thank all the members of the Distributed Information Systems Lab for their support.

I am also thankful to the Systems Staff at the School of Computer Science, including Ron Simpson, Andrew Bogecho and Kailesh Mussai for their help and co-operation from time to time.

Finally, I would like to thank my parents, who have been a constant source of inspiration and encouragement and have always been extremely supportive of all my academic endeavours.

Abstract

An exponential growth in internet usage and penetration amongst the general population has led to an ever increasing demand for e-commerce applications and other internet-based services. E-commerce applications must provide high levels of service that include reliability, low response times and scalability. Most e-commerce applications follow a multi-tier architecture. As they are highly dynamic and data-intensive, the database is often a bottleneck in the whole system as most systems deploy multiple application servers in the replicated application tier, while only deploying a single database as managing a replicated database is not a trivial task. Hence, in order to achieve scalability, caching of data at the application server is an attractive option.

In this thesis, we develop effective load balancing and caching strategies for read-only transaction workloads that help scaling multi-tier architectures and improve their performance. Our strategies have several special features. Firstly, our strategies take into account statistics about the *objects* of the cache, such as access frequency. Secondly, our algorithms that generate the strategies, despite being object-aware, are generic in nature, and thus, not limited to any specific type of applications. The main objective is to direct a request to an appropriate application server so that there is a high probability that the objects required to serve that request can be accessed from the cache, avoiding a database access. We have developed a whole suite of strategies, which differ in the way they assign objects and requests to application servers. We use distributed caching so as to make better utilization of the aggregate cache capacity of the application servers. Experimental results show that our strategies are promising and help to improve performance.

Résumé

Une croissance exponentielle de l'utilisation d'Internet et sa pénétration dans la population générale ont conduit à une demande toujours croissante d'applications de commerce électronique et d'autres services basés sur l'internet. Les applications de commerce électronique doivent fournir des niveaux élevés de services qui comprennent la fiabilité, un court temps de réponse et de la variabilité dimensionnelle. La plupart des applications de commerce électronique suivent une architecture multi-niveau. Comme elles sont très dynamiques et possèdent une forte intensité de données, la base de données est souvent un goulot d'étranglement dans le système en entier comme la plupart des systèmes déploient des serveurs d'applications multiples dans l'application tierce reproduite. D'un autre côté, le déploiement d'une base de données unique pour la gestion d'une base de données répliquée n'est pas une tâche simple. Ainsi, afin de parvenir à une variabilité dimensionnelle, la mise en cache des données au serveur d'applications est une option attrayante.

Dans cette thèse, nous développons un équilibrage de charge efficace et des stratégies de mise en cache qui aident à échelonner les architectures multi-niveaux et à améliorer leurs performances. Nos stratégies ont plusieurs caractéristiques particulières. Premièrement, nos stratégies prennent en compte les statistiques sur les objets de la mémoire cache, comme la fréquence d'accès. Deuxièmement, nos algorithmes qui génèrent les stratégies, tout en étant conscients des objets, sont de nature générique, et donc, ne se limitent pas à un type spécifique d'applications. L'objectif principal est de diriger une requête au serveur d'applications approprié afin qu'il y ait une forte probabilité que les objets requis pour servir cette demande puissent être consultés à partir de la mémoire cache, évitant un accès à la base de données. Nous avons développé toute une série de stratégies qui diffèrent dans leur façon d’assigner des objets et des requêtes aux serveurs d'applications. Nous utilisons une mise en cache distribuée de manière à mieux utiliser la capacité totale de la mémoire cache des serveurs d'applications. Les résultats expérimentaux montrent que nos stratégies sont prometteuses et permettent d'améliorer les performances.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Electronic Commerce (or *e-commerce*) can be defined as the buying or selling of goods over the internet. With the explosive growth in internet usage across the world in the last decade, e-commerce has become one of the most important and rapidly growing internet-based activities. Forrester Research [33] predicts that online retail sales in the U.S. will reach $278.9 billion in 2015. Over the years, with increase in internet usage and improvement in technology, the face of e-commerce has changed from simply being a process for executing commercial transactions electronically by Electronic Funds Transfer (EFT) to being a complete store-like retail experience for the consumer today. Service providers nowadays are bound to provide a guaranteed level of service to their customers. Given the market-size and the dependence of businesses on e-commerce, it has been essential to develop e-commerce applications which are highly available, scalable and have low response times. Hence, the architecture of e-commerce systems, their flexibility to change with change in business needs and increase in consumer growth and their ease-to-use has come under close scrutiny over the past few years.

Keeping into consideration these demands of e-commerce businesses, applications that follow a multi-tier architecture approach have become the norm. A multi-tier architecture basically splits up the entire system into three separate components: the client tier, the application tier and the data tier. The client tier is the point of interaction for the users with the system. The application tier consists of the business logic which is at the core of the e-commerce system. Finally, the data tier is the tier where all the data related to the application is stored. Since the components are separated, it is easier to develop, maintain and scale such systems.

In order to scale multi-tiered architectures, we can increase the number of resources present at any tier either by augmenting the hardware at each machine or by replication of a tier. Replication is often the preferred approach, since it is cheaper and improves availability of the system. If we are using replication, usually a load balancer is placed in front of the replicated tier which acts as a point of contact for the tier preceding it. Since a distributed database is difficult to manage, replication of the data tier is not as common as that of the application tier. Thus, with a single database the database tier more often than not ends up being the bottleneck in the system. This is especially true for e-commerce applications which are dynamic and data-driven. As the total number of users of the system increase, there is a tremendous increase in the size and the load on the database layer.

In order to lower the load on the database, caching of the data at the application tier is a popular solution. Data stored in the cache can be accessed very fast, and thus, it helps to improve performance since requests for the cached data need not go to the database and can be served from the cache itself in an efficient manner. To summarize, advantage of cached data is two-fold: access is extremely fast, and load on the database is reduced.

In this thesis, we examine how intelligent load balancing and caching can improve performance in multi-tier architectures and make them scale better. We analyze the cache usage statistics of the application server by the means of logging, and develop several different load balancing and caching strategies for the application server tier. A monitoring tool has been specially created for analyzing access logs and for subsequent generation of load balancing and caching strategies. The monitoring tool runs on a server independent of the machines in any tier. A key characteristic of the developed strategies is that they are developed primarily based on the statistics of the *objects* that are recorded to be accessed at the cache.

The load balancing and caching strategies are closely coupled together and are constructed with the global objective of directing a request to an application server which is most likely to have the data required by the request in its own cache, i.e. increasing the cache hit rate. The *load balancing strategy* consists of instructions to the load balancer where to direct the requests coming from the clients. The *caching strategy* contains instructions to the application servers regarding which objects to cache and for how long to cache them. An important feature of our strategies is that they also explore how placement of cache objects impacts performance. For example, we investigate how replication of cache objects affects performance of the system.

Moreover, we collect statistics which are then analyzed in real-time (i.e. when the system is currently running), and create load balancing and caching strategies on the fly. Also, the monitoring tool allows for great flexibility, giving us the choice of configuring the interval of strategy generation. Since these strategies are basically an outcome of the analysis of the cache usage statistics at the application server, our solution is *application-independent*.

Further, we use an existing distributed co-operative caching infrastructure [4], which increases the aggregate cache capacity of the system. It allows us to fetch requested data from the cache of other application server in addition to the cache of the server where the request has arrived. We evaluate the performance of our strategies using a well-known e-commerce benchmark. The results show that there is a significant potential of improving performance with effective load balancing and caching strategies.

1.1 Contribution

The following are the main contributions of this thesis:

- Development of scalable caching and load balancing strategies for read-only transactions in multi-tier architectures primarily based on the characteristics of the objects used in the application

- Development of a monitoring tool as the necessary infrastructure for generation of these strategies

- A detailed analysis and evaluation of these strategies based upon various parameters and in different scenarios

1.2 Thesis Outline

This thesis is divided into 6 chapters, including this introductory chapter. In Chapter 2, we give background information on topics concerning our thesis such as multi-tier architectures, scalability, load balancing, caching, and a distributed caching framework [4] that we are utilizing in this thesis. We also discuss related work. Chapter 3 describes the architecture and the working of the monitoring tool that we have developed in order to generate our caching and load balancing strategies. We explain in detail our caching and load balancing algorithms in Chapter 4, which is the main contribution of this thesis. Experimental results and evaluation of strategies are discussed in Chapter 5. Finally, in Chapter 6, we put forth our conclusions and some possible directions for future work on this thesis.

Chapter 2

Background

2.1 Multi-Tier Architectures

Initially, when computers came into use, there were mostly mainframes wherein both the application and the database were co-located in a single machine. Clients accessed this machine using minimal functionality terminals. This model was followed by the popular two-tier client-server architecture, where the client and the server could be located on different networks. In a thin client/fat server model, the client just contains the user interface while the business logic and the database are located at the server side. On the other hand, in a fat client/thin server model, the client also hosts the business logic along with the user interface. With the exponential use in web-based applications and services, the client-server architecture is no longer a scalable model. A multi-tier architecture is essentially a repeated application of the client-server model, in which one of the middle tiers is a client for the immediate lower tier and a server to the immediate higher tier.

In the three-tier architecture, there are the following three tiers:

- The client tier, which contains the user interface.
- The application tier, which contains the business logic that performs all the

processing.

- The data tier where the database servers lie and all the information is stored.

A three-tier architecture can be visualised as shown in Figure 2-1.

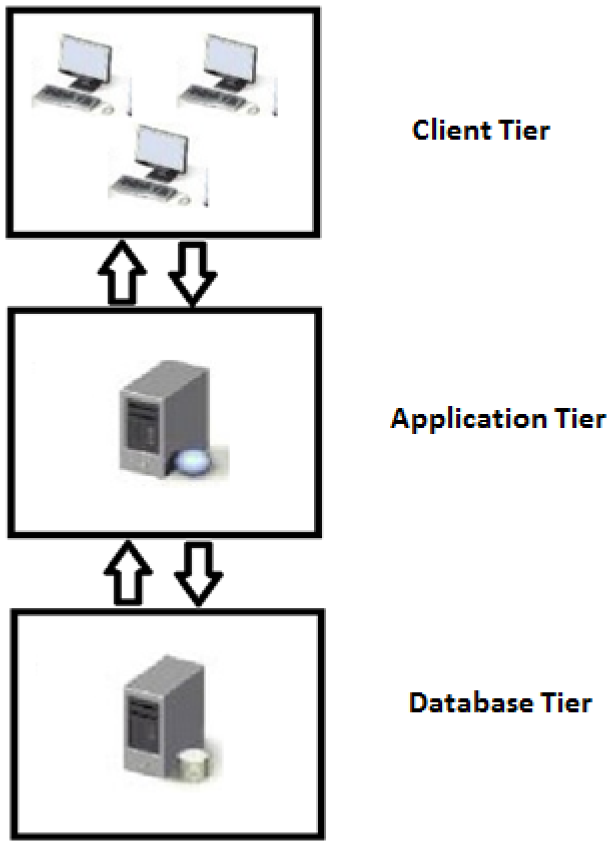

Figure 2-1 Three-tier architecture

Each tier can be deployed on different platforms independent of each other. Having a three-tier architecture makes it easier to make changes or replace any tier without affecting the other tiers. With growth in internet usage, data requirements have also increased exponentially and hence, with a separate database tier, data can be managed efficiently independent of the other tiers resulting in greater scalability.

2.2 Scalability

Scalability refers to the capability of the system to increase total throughput under increasing load along with increase in resources available. A scalable application is one which is able to increase throughput in proportion to the increase in load on the system and the corresponding increase in resources to serve them.

Scalability may be achieved by one of the two following methods:

- Scaling horizontally, in which we add new nodes to a system. For example, in the context of a multi-tier architecture, we may add multiple application servers to the application tier or add database servers to the database tier. A drawback of this approach is that it leads to increased management complexity amongst the different nodes in a given tier.

- Scaling vertically, where we try to increase resources such as number of CPUs and memory available to a single node in the system. Though simpler to manage, adding hardware to an existing system is highly expensive. Moreover, vertical scaling leads to all the resources being located on a single machine. If that machine is unavailable, the entire system becomes unavailable. Hence, though scalability may be achieved, availability is not addressed by this approach.

Due to its properties, scaling horizontally is preferred for the application tier. However, if we have multiple application servers, the client has to be aware of the addresses of each of them. In order to avoid this resulting loss of transparency, load balancers are usually placed in front of the application tier and subsequently all requests to that tier must be routed through that load balancer. Thus, the load balancer distributes the incoming requests amongst the different servers available in a given tier. It is also possible to horizontally scale the database tier, having multiple machines at the database layer. However, this results into a distributed database, which is very complex to manage, and hence is not used in this thesis.

In Figure 2-2, we can see how a load balancer in front of the application tier distributes incoming requests from a client in a three-tier architecture. In this thesis, all experiments conducted follow this model.

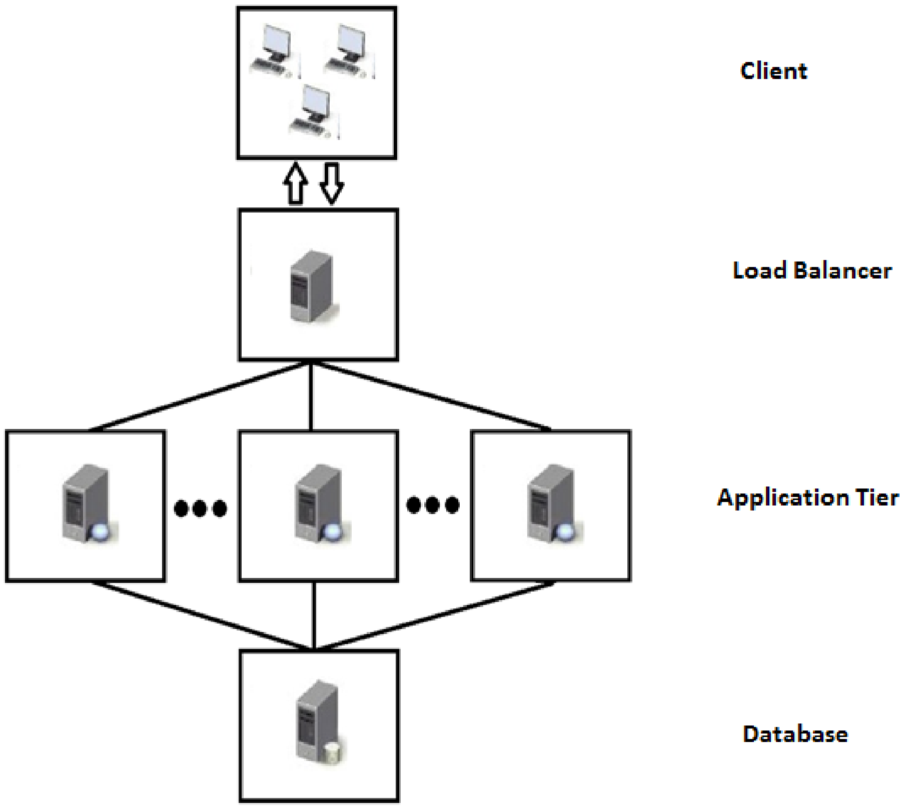

Figure 2-2 Three-tier architecture with horizontally scaled application tier

2.2 Load Balancing

As shown in the previous section, load balancing is used to distribute the incoming workload across multiple servers. A load-balancing solution usually has two components; the entity performing the load balancing and the algorithm which it uses to do it. The method of load balancing can have a significant impact on the performance of the system.

Cardellini et al. [1] classify load balancing solutions into four main approaches based on the entities that perform load balancing:

- Client Based Approach
  In this approach, web clients themselves are aware about the replicated servers and route the request accordingly. Clearly, this approach is not scalable because as the number of web servers increase or change, it becomes increasingly difficult to store and maintain information about them at the web clients.

- DNS Based Approach
  A DNS server is used to map IP addresses to their respective domain names. In the DNS based load balancing approach, load balancing can be achieved by mapping more than one IP address to a domain name. The DNS server then chooses the IP address based on some load balancing algorithm.

- Dispatcher Based Approach
  The dispatcher based approach provides more control over load balancing by having a node in the system act as a dispatcher. This method is also transparent to the client since the dispatcher has a unique and virtual IP

address. The dispatcher is aware of the servers in the tier and identifies them through private addresses. Dispatcher based approaches can use a variety of load balancing algorithms.

- Server Based Approach

  This approach uses a two-level dispatching mechanism. Initially, a server is chosen using a DNS server. Following this step, this server chooses another server to serve the request. Thus, all servers have a chance to participate in the load balancing process.

In this thesis, we work with the dispatcher based method.

Load balancing algorithms usually fall into two categories:

- Content aware load balancing algorithms

  In these algorithms, a server is chosen based on the information contained in the request sent by the client. Custom algorithms can be implemented in this approach to exploit the knowledge contained in the request to the fullest.

- Content unaware load balancing algorithms

  In these algorithms, a server is chosen without any consideration to the content of the request sent by the client. Examples of this approach include random choice, round robin, weighted round robin, sending the request to the server with the least number of outstanding requests, etc.

2.4 Caching in JBoss Server

In this thesis, we use JBoss Application Server (JBoss AS) by Red Hat as the application server. JBoss AS is an open-source Java EE-based application server. From JBoss AS 4.0 and upwards, Hibernate integration support is provided [2]. Hibernate is a well-known object persistence framework developed by the JBoss community. It is an object-relational mapping (ORM) library for Java, providing a framework for converting Java objects to tables in relational databases and vice versa.

In computer systems, a cache is an area of high speed memory that is used to store frequently accessed data. If requested data is contained in the cache (also known as a *cache hit*), this request can be served by simply reading the cache. If it is not present in the cache (also known as *cache miss*), the data has to be recomputed or fetched from its original storage location such as a database. A cache miss is comparatively slower than a cache hit. The percentage of accesses that result in cache hits is known as the *hit rate* of the cache. A higher hit rate means more requests being served from the cache, leading to faster performance. Thus, caching is used to improve system performance; however it is essential that its usage and configuration parameters such as size are adjusted depending on the type of application so as to make optimal use of it. For example, for applications wherein the same data is requested frequently, it can improve performance to a great extent. For applications where the data requested varies a lot or there are frequent changes to the data, it may not improve performance greatly. This is because the cache has a fixed capacity, and as data changes frequently, cached data is no longer valid and cannot be used.

In order to improve performance, Hibernate uses caching [3]. This caching occurs at chiefly two levels:

- First-level Cache

  The first-level cache in Hibernate is a session-level cache. A session in Hibernate is a unit of work that groups data access operations. Hibernate uses this cache by default. In the course of a session, Hibernate stores objects and queries corresponding to that session in this cache. At the end of the session, the contents of the cache are neither accessible nor valid, and are destroyed.

- Second-level Cache

  The second-level cache in Hibernate has an application-wide scope and one can plug in any cache that implements a well-defined interface. All requests have access to this second-level cache. In our implementation, we use Ehcache [5] as the second level cache.

Figure 2-3 shows the organization of the two-level cache in Hibernate.

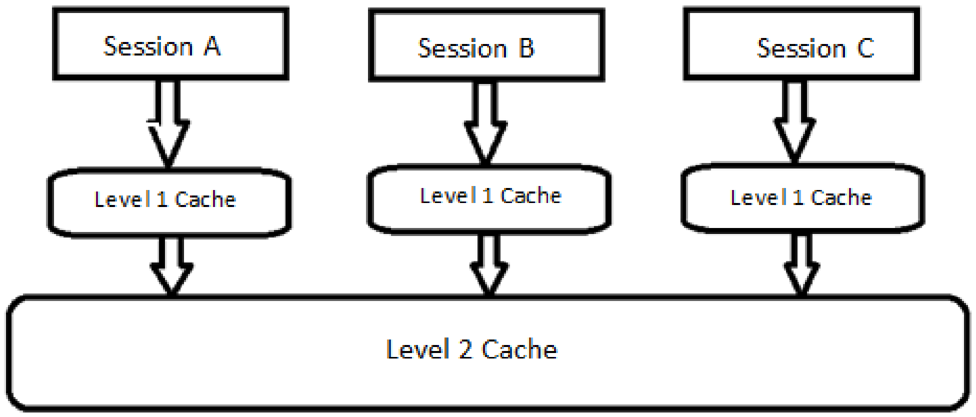

Figure 2-3 Two level cache in Hibernate

In addition to these two levels, Hibernate also uses a query cache, which is closely integrated with the second-level cache. The query cache is used to store the result set of a query made against the database. The key of the query cache contains the query string itself and any other parameters passed with the query. The value consists of just the primary key identifiers for all of the entities returned by the query. When a query is made that hits the query cache, that set of entity identifiers can be retrieved and then resolved through the first or second level caches instead of retrieving those entities from the database.

Elements in the cache can be removed by eviction due to memory constraints or by invalidation, wherein the cached element is removed since it is no longer relevant due to passage of time or changes to the copy of the data at the database. The removal in the second-level cache and the query cache is independent of each other. For example, if an element from the query cache is removed, only the query string and the object identifiers are removed, the corresponding elements in the second-level cache remain intact. The same is true for an element removed from the first or second-level cache; the corresponding elements in the query cache remain unaffected.

Consistency of the cache is also an issue which needs to be managed well. In order to ensure that the request is served with the correct data, it is important to keep the contents of the cache consistent with that of the database. Also, since we have a replicated application tier, it is possible that the same element is replicated across the caches at multiple sites. In this case too we need a protocol to ensure that the copies of the cache remain consistent at all the sites. There are several cache consistency mechanisms that can be used such as TTL (time-to-live) fields, client polling and invalidation protocols. [31] and [32] present comparative studies about these different methods.

In this thesis we do not deal with write operations to the database; hence we do not implement a consistency mechanism for the cache. However, we could plugin one of the above mentioned consistency mechanisms into our solutions to ensure cache consistency.

2.5 Distributed Caching Infrastructure

As described earlier, we use horizontal scaling for the application tier, which results in having multiple application servers. Each application server has its own cache and so we have multiple caches present at the application tier. In order to utilise the combined capacity of the caches, we make use of a distributed co-operative cache as described in [4]. An architectural diagram of the same is shown in Figure 2-4. It uses JBoss AS as the application server, Hibernate for ORM, Ehcache [5] as the second-level cache for Hibernate and PostgreSQL [25] as the database. All the communication that takes place between the application servers to maintain this distributed caching infrastructure is done using JGroups [34], which is a reliable multicast system written in Java.

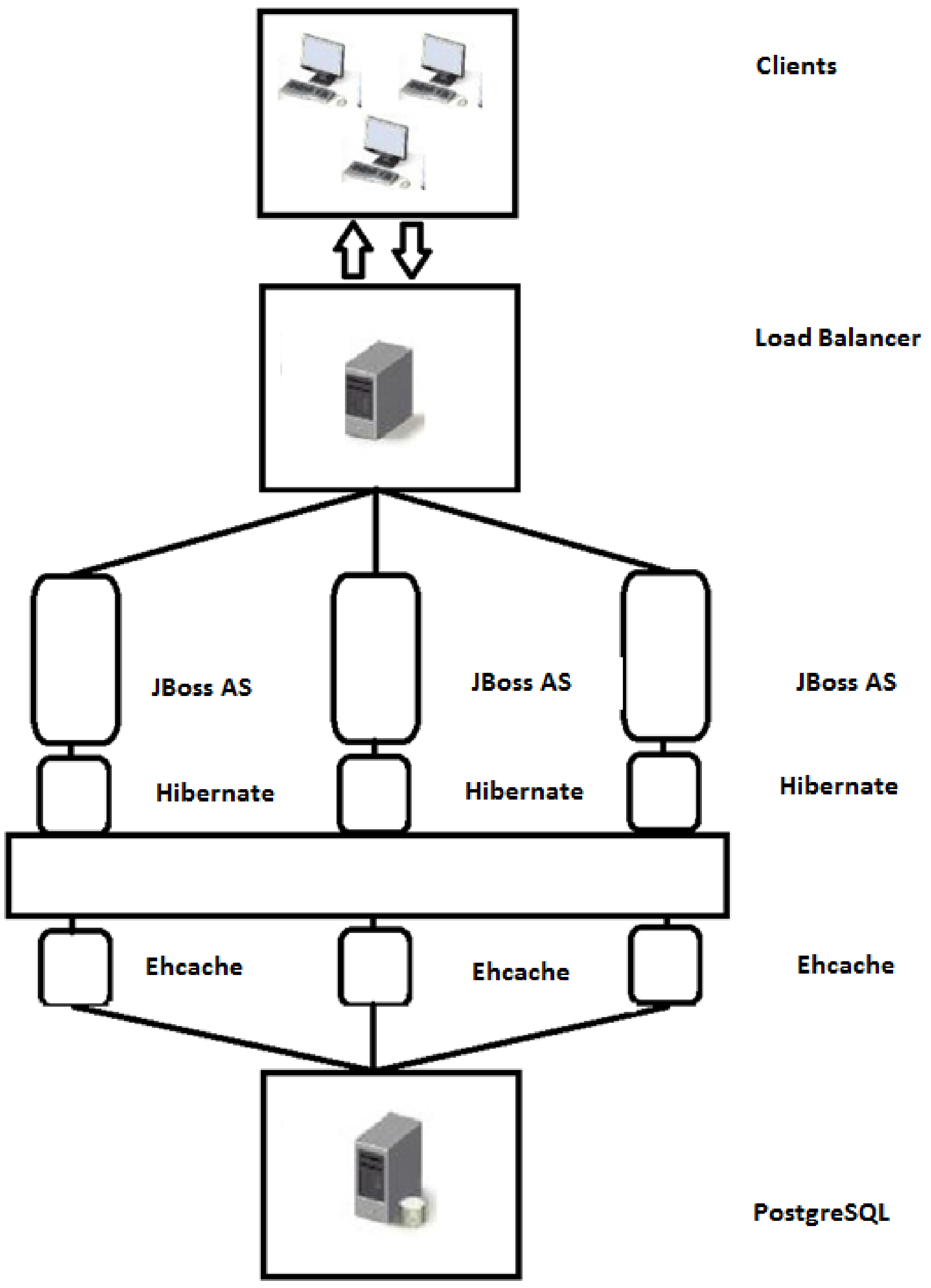

Figure 2-4 Distributed co-operative cache

In this system, each application server has knowledge about the contents of the caches of all application servers and can access these contents. Thus, we have a larger effective cache capacity and hence, the load on the database decreases considerably, improving performance. As an illustration, if a client requests a query or object to an application server that does not have that query or object in its own cache it can check if other caches have that requested query or object. If it is present at one of the other servers, a remote call is made to that cache to retrieve it from there.

At each application server, the co-operative cache maintains a cache directory for a look-up of the cache objects' location. For each object lying in any of the caches, the directory contains a reference to the location of that object. Each entry in this directory looks like:

Object Key ↔ Location of Object

In order for the directory information to be consistent and up-to-date, information about an object being added/removed from the cache is multicast to all the caches present in the tier.

In the distributed co-operative cache, a given object can be stored only at a single location at a time. This means that if a request arrives at a given application server, it checks if the objects required to satisfy the request are present in any of the caches. If they are present at a remote location, they are fetched from the remote site. If not, the local cache is checked and if present they are fetched from the local cache. If this fails, they are loaded from the database. However, in order to implement the various caching strategies described later in this thesis, we have modified this distributed co-operative cache so that it is possible for a given object to be present at more than one location at a time. In this case, a directory entry looks like:

Object Key ↔ <List of Locations of the Object>

Thus, replication of objects in the cache may be possible.

2.6 Related Work

Several different approaches have been explored for content aware load balancing and replication of popular data items. Both of these strategies help to improve performance by increasing the cache hit rate.

Elnikety et al [6], put forward a memory-aware load balancing (MALB) technique to dispatch transactions to replicas in a replicated database. They show their technique performs better than traditional algorithms such as round-robin and least connections for dynamic-content workloads. The reason for the improvement is that their load balancer utilises information about the size and the contents of the working set of transactions to assign them to replicas in such a way that the data items accessed can be cached in main memory. However, in this case the load balancer lies in front of the database replicas, and not in front of the application server as in our case and in [7]. In [7], a cache aware load balancing solution is presented which makes effective usage of the second-Level cache of Hibernate in the application server. With the help of intelligent load balancing techniques, an effort was made to distribute each request to an application server which could possibly have the data needed for this request in its own cache. The algorithm used for this is inspired from the concepts used in [6]. However, this load balancing solution does not have a corresponding cache assignment scheme and is application-dependent.

Cherkasova L. [8] proposed a locality aware solution called FLEX for a cluster hosting several websites, mainly applicable for web content hosting services. FLEX allocates sites to different machines in a cluster based on their popularity. The load balancing strategies that we produce utilise the information about the popularity of the objects, and tend to distribute them across the application servers. HACC [9] distributes incoming requests to a cluster of web servers to nodes in such a way that the locality of reference that occurs on individual nodes in the cluster is maximized, which helps in improving performance. Our load balancing strategies also share this objective of directing a request to an application server where the request can be served from the local cache, without having to access the database.

A strategy to replicate very popular data items is presented in [10]. An algorithm known as workload-aware request distribution strategy (WARD) is proposed, which assigns a small set of most frequently accessed files to each node in a cluster, while distributing the rest of them across the different nodes in the cluster. In addition, they also describe an algorithm to compute the ideal size of these replicated files. Similarly, in this thesis, we try to exploit the popularity of data items, and some of the algorithms replicate the most popular objects across the application servers.

Although maintaining and serving web documents is a different problem and is not directly related to the work in this thesis, certain concepts regarding replication and caching of web documents are useful in our scenario, too. For our context, web documents may be thought of as cache objects. In [11], the need for dynamic selection of caching and replication strategies for web documents is shown. Pierre et al [12] illustrate that there is no single way of caching or replicating documents efficiently and that each document should be replicated with a custom policy. We try to make use of these concepts for our caching and load balancing strategies, wherein we produce our load balancing and caching strategies in real-time, and also make different policies for each object.

Issues pertaining to scalability of multi-tier architectures have also been investigated previously. Consistency of data is an area which has been thoroughly worked upon.

Management of multiple replicated tiers is discussed in [13]. Replication solutions for each tier are examined independently and then strategies to combine the replicated tiers are presented. In [14], an approach that takes care of both availability and scalability for multi-tier J2EE applications is presented. The authors propose a replicated multi-version cache that takes care of consistency, availability and scalability and also takes into account both the application-tier and the database-tier. [15] puts forward different mechanisms for providing consistency in dynamic web applications. In [16], the authors deal with reliability in three-tier architectures. They provide a specification of an end-to-end reliability contract in the context of a three-tier architecture. [17] analyses the effects of query caching on multi-tier architectures with a replicated application-tier. However, none of this work on multi-tier architectures has its focus on load balancing or cache assignment.

The concept of co-operative caching and replication of data has also been explored in the context of peer-to-peer systems. In [18], the authors propose a co-operative caching scheme for XML documents. Replication strategies in unstructured peer-to-peer systems are analysed in [19]. Co-operative cache is also used for full text document search in peer-to-peer systems in [20]. A decentralized peer-to-peer web cache called Squirrel is introduced in [21] which basically allow web browsers to share their local caches.

Memcached [30] is an open source distributed memory object caching system developed by Danga Interactive. However, in our thesis, we use the distributed co-operative cache, since it is easier to implement the concepts of replication of objects on top of it compared to Memcached. Also, we want to have better control on where to put certain objects in the cache, which would be difficult if we were to use Memcached.

Chapter 3

Monitoring Tool

The caching and load balancing strategies, which are the main contribution of this thesis, are developed by statistical analysis of the logs generated at the application servers when the application servers are running. There are mainly two tasks involved in this process: generating logs, and analysing these logs. A monitoring tool has been developed in collaboration with Rizwan Maredia [35] to take care of these two tasks. This monitoring tool has two components. The log capture component of the tool at the application servers take care of generating the logs, while the log processing and analysis component, which can be located on a separate server, receives and analyses the logs.

A component diagram of the monitoring tool is shown in Figure 3-1.

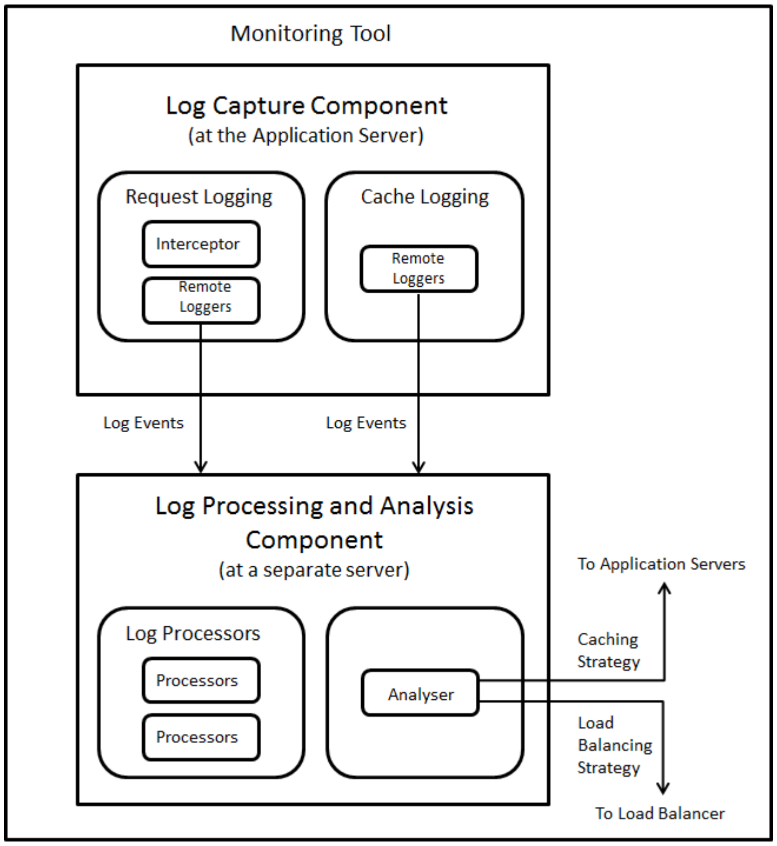

Figure 3-1 Monitoring Tool

3.1 Log Capture Component

The log capture component at the application server does mainly the following two functions:

- intercept incoming requests
- log and forward the logs to the log processing and analysis component

We place an interceptor at the application server, which intercepts requests coming in from the client via the load balancer. Information contained in this request that is important to our analysis server is extracted and subsequently sent to the log processing and analysis component.

We call the logs referring to the request as *access logs*. Furthermore, we log all calls to the second-level cache of Hibernate. This includes calls to fetch an object from the cache or put an object into the cache. This enables us to record the access frequency and other metrics of each cache object. We call this information a *cache log*.

Logs are sent immediately to the log processing and analysis component. The log processing and analysis component is typically running on a different machine to not interfere with the application processing. Therefore, we refer to our approach as *remote logging*. The analysis tool is fed with the logging data in real-time, as it is produced, providing for dynamic load balancing and caching.

Since the logs are directly sent to the log processing and analysis component from the various application servers, they may arrive out of order. This makes it impossible to determine a relationship between a request (tracked through the access logs) and the cache objects it accesses (tracked through the cache logs). But such information is crucial to the analysis. In order to address this issue, we introduce a Globally Unique Identifer (GUID) for each request to be inserted into all the logs, including cache logs that are triggered by this request. This ensures that we have the mappings between the requests and the cache objects it accesses. More details about the GUID implementation are given in Section 5.5.

3.2 Log Processing and Analysis Component

Log processing and analysis are the most important tasks of the monitoring tool. In our implementation, both these tasks are done on the same machine. This is efficient and desirable since the information statistics computed from the logs are used in our analysis to decide the appropriate load balancing and caching strategy.

3.2.1 Log Processing

The log processing and analysis component listens on a specified port for logging events that come from the log capture component at the application server. The remote loggers at the application servers send the access logs and the cache logs to the log processing and analysis component as described in the previous section.

Statistics Manager

When the log processing and analysis component starts, it instantiates a statistics manager which in turn loads different components depending on parameters defined in a configuration file. These components are able to communicate with each other as they all share this manager. The statistics manager receives log events as they arrive at the log processing and analysis component.

The statistics manager sends the log message to each of the log processors (see Log Processors section) which have registered with it for further processing.

Log Processors

The main tasks that the log processors do are to generate useful statistics from the logs that could be used in our analysis to generate strategies for caching and load balancing. Examples of this include determining access frequency of a cache object, frequency of a HTTP request and mapping a HTTP request with the corresponding cache objects it accesses. The log processors maintain separate data structures (and hence separate statistics) for each application server.

In line with the above mentioned tasks, we have the following log processors:

A. Cache Log Processor

   The cache log processor processes the cache logs. It maintains a list of *Cache_Object*s, known as *cacheObjectList*. A *Cache_Object* is a data structure which contains the following information:

   - Object key, which serves as a unique identifier.

   - A counter which maintains the count of how many times this object has been accessed at this particular application server.

B. HTTP Request Log Processor

   The HTTP request log processor processes the access logs. It maintains a list of *HTTP_Request_Object*s. An *HTTP_Request_Object* is a data structure which contains the following information:

   - The URL of the request, which serves as a unique identifier.

   - A counter which maintains the count of how many times this request has arrived at this particular server.

C. Request - Resource Mapping Processor

The request-resource mapping processor keeps a track of both the cache and the access logs. Utilising the information contained in both logs, it maintains two maps:

- *Request_to_Resource_Map*, which maps HTTP requests to the cache objects it has accessed.

- *Resource_to_Request_Map*, which maps a *Cache_Object* to the HTTP requests that have accessed that object.

All those log processors that have registered with the log processing and analysis component receive logging events from it. The log processors read the log and parse the message for extracting information from it and store it into their respective data structures. These data structures are passed on to the analyser for further processing.

3.2.2 Analyser

The analyser is the brain of the entire monitoring tool - it runs the algorithm which generates the caching and load balancing strategies using the statistics produced by the log processors. In this thesis, we have developed several algorithms, each producing a different load balancing and caching strategy.

The analyser runs at periodic intervals of time. The desired interval can be read from a configuration file. The algorithms which can be run in the analyser are described in detail in the next chapter. The period from which we want to use the statistics can also be specified. For example, we can specify that we want to use the statistics computed from the last 15 minutes only for our analysis.

The analyser takes the data structures generated by the log processors as inputs, and produces caching and load balancing strategies as its output. At the end of the run of the analyser, a unique caching strategy is produced for each application server, called *ASPolicy* (since it is produced for the application server) and is sent to the application server. The analysis also produces a load balancing strategy, called *LBPolicy* (since it is produced for the load balancer) to be sent to the load balancer. After the strategies are received at the respective sites, the application servers use the ASPolicy to make caching decisions for objects, and the load balancer uses the LBPolicy to make load balancing decisions.

ASPolicy

The ASPolicy is a set of entries of the form:

$$\textit{Object\_Key} \leftrightarrow \textit{Rule}$$

Here each *Object_Key* is the unique identifier of an object, and *Rule* is a data structure that contains a set of instructions that may be applied to that object. When an object is fetched from or put into the cache at the application server, the application server checks ASPolicy to see if there is a rule to be applied for this particular object. *Rule* consists of a *Rule_Type* which can be to *store* or *move* a given cache object, a Time-To-Live (TTL) to be specified for that object, and a *Destination_Server_List*, if the *Rule_Type* is *move*. The *Rule_Type store* is for instructing the application server receiving that rule to store the object at the cache at that site with a TTL specified in the *Rule*. The *Rule_Type move* is for instructing the application server receiving that rule to move the object to o the application servers listed in *Destination_Server_List*.

We have divided TTL into three categories, which can ideally be assigned to objects in decreasing order of their popularity:

- STABLE; objects having STABLE TTL have the greatest TTL value.
- LONG; objects having LONG TTL have a medium TTL value.
- SHORT; objects having SHORT TTL have the least TTL value.

If an object which is accessed at the application server is not cached anywhere and is not present in the ASPolicy, it is simply stored with a SHORT TTL.

Creating separate categories for TTL values provides a ranking scheme for cache objects. For example, when the cache is full, and an object must be evicted to make way for a new object to be inserted into the cache, the object with the greatest *evictionPotential* (where *evictionPotential* is calculated as *last accessed time* minus *TTL value*) will be chosen as an eviction candidate. Thus, the higher the TTL, the lesser the chances are that the object will be evicted from the cache. In this way we can ensure that the most popular objects are the least likely to be evicted from the cache.

Although we do offer a *move* rule, we have not used it in our strategies. Upon receiving a *move,* an application server has to send the object to another application server. This creates additional communication between the application servers and it would also result in additional overhead related to the maintenance of the directory that is a part of the distributed co-operative cache, as described in Section 2.4. Instead, the effect of a *move* can be achieved by the use of two *store* rules for the same object, but at different application servers. We can set the TTL for the object to be SHORT for the server which should be releasing the object, and setting the TTL at the server receiving the object to be STABLE/LONG.

LBPolicy

The LBPolicy is a set of entries of the form:

HTTP_Request_URL ↔ IPAddresses

Here for each HTTP request, its URL is mapped to a set of IP Addresses of application servers (each application server has a private address known to the load balancer). When a request arrives at the load balancer, it checks LBPolicy to see if there is an entry for that request. If there is an entry for that request, the load balancer chooses an IP address at random from the set and sends the request to that application server. If the request does not have an entry in LBPolicy it is simply directed to the application server chosen by the default load balancing algorithm running at the load balancer (for e.g., round robin).

3.3 Request Flow

We have seen the architecture and the functions of the various components of the monitoring tool. Now, let us examine how the presence of the monitoring tool changes the data flow in a single request.

In absence of the monitoring tool, the request flow is as shown in Figure 3-2.

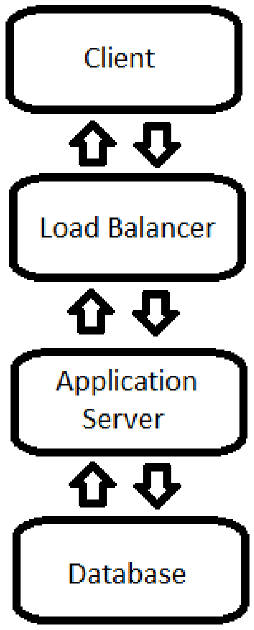

Figure 3-2 Request flow in absence of monitoring tool

The client submits the request, which goes to the load balancer. The load balancer selects an application server by its own load balancing algorithm, where the request is sent. The application server does not request the database for objects if the objects required are in its own cache or a remote cache. Else, it requests objects from the database, puts the object in the local cache and the response to the client follows the reverse path.

In presence of the monitoring tool, the request flow looks like as shown in Figure 3-3. The difference in this case is at both the load balancer and the application server. When the request arrives at the load balancer, the load balancer checks if there is a LBPolicy present, and if it is present, if the URL of the request has an entry in the LBPolicy, it sends the request to that application server. At the application server, if there is a rule present for the cache objects required to serve the request, it is applied. The logs generated are sent to the log processing and analysis component.

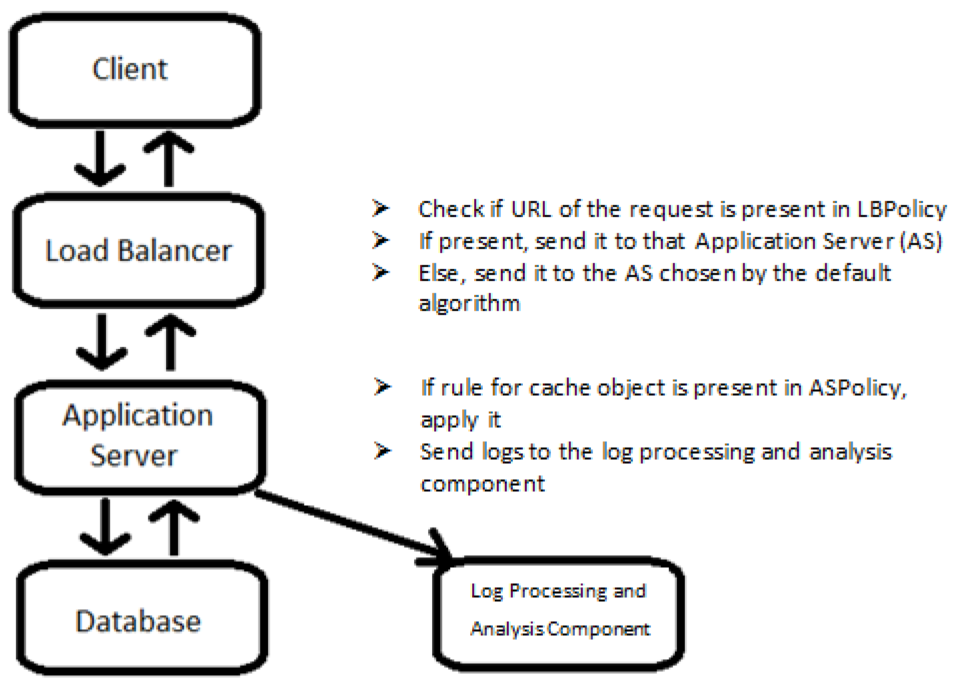
Client
Load Balancer
Application Server
Database
Log Processing and Analysis Component
Check if URL of the request is present in LBPolicy
If present, send it to that Application Server (AS)
Else, send it to the AS chosen by the default algorithm
If rule for cache object is present in ASPolicy, apply it
Send logs to the log processing and analysis component

Figure 3-3 Request flow in presence of monitoring tool

Chapter 4

Object Based Load Balancing and Caching Strategies

4.1 Introduction

In this thesis, we focus on developing caching and load balancing strategies primarily based on the statistics of the objects of the cache which have been generated by the log processors. The load balancing strategy, to a great degree, is closely tied to the caching strategy. Our ultimate goal is to develop a holistic caching solution, wherein there is a great chance that the request sent to an application server by the load balancer can be served from the cache of that application server only or at least the cache of another application server, minimizing requests sent to the database, which more often than not becomes a bottleneck in the system. Hence, the load balancing strategies we develop in this thesis are cache-aware.

The caching strategies developed explore both replication and distribution of cache objects across application servers. An important characteristic of the algorithms is that they are dynamic, and hence they can be adapted to change in workload characteristics. Moreover, they are also application – independent. Thus, although we have performed experiments in this thesis on the RUBiS benchmark, they could be used for any application without any changes. This chapter describes in detail the algorithms we have developed for use in the analyser in the log processing and analysis component.

In Ehcache [5], the cache size can be specified in terms of the number of objects it can hold. The caching strategies that our algorithms produce create a cache assignment policy that essentially divides the total cache available at a given application server into three parts:

a) One part of the cache is reserved for objects replicated at each server. These objects are the ones that have been identified during our analysis as the most popular ones and have been added to the ASPolicy typically with a STABLE TTL. We denote the number of replicated objects in the cache as *REPLICATION_COUNT*.

b) The second part of the cache is intended for objects that are distributed across the application servers. These objects have also been added to the ASPolicy with a STABLE/LONG TTL. We denote the number of distributed objects as *DISTRIBUTION_COUNT*.

c) The remaining part of the cache acts as the default distributed co-operative cache would. The objects present in this part are not present in the ASPolicy and are stored with a SHORT TTL. Thus, if the total capacity of the cache is *TOTAL_COUNT* objects, the number of objects that this cache can hold is effectively *TOTAL_COUNT – (REPLICATION_COUNT + DISTRIBUTION_COUNT)*. We denote this by *DEFAULT_COUNT*.

Finally, given the above parameters, if the number of servers is denoted by *SERVER_COUNT*, then the aggregate cache capacity is *REPLICATION_COUNT + SERVER_COUNT * (DISTRIBUTION_COUNT + DEFAULT_COUNT).* Since at each site the part of the cache which stores the replicated objects contain identical objects, as compared to the distributed co-operative cache, our aggregate cache capacity is (*SERVER_COUNT* -1) * *REPLICATION_COUNT* times lesser. Also, if our strategies would not incorporate replication and/or distribution of objects, then *REPLICATION_COUNT* = 0 and/or *DISTRIBUTION_COUNT* = 0 respectively.

The algorithms mainly use the characteristics and properties of the cache objects; however, we also make use of the information about the requests. This is made possible by the request–resource mapping log processor which is described in the previous chapter. Using that log processor, we are able to determine the requests which have accessed any given cache object. Thus, based on whether or not the algorithms use this information about requests, we can divide them into two categories:

- Request-unaware algorithms, which do not utilize information about HTTP requests

- Request-aware algorithms, which utilize information about HTTP requests that access the objects from the request–resource mapping log processor

At any given time, the analyser can run only one algorithm. The possible algorithms are described in the rest of the sections in this chapter.

Some common data structures and notations which are used across the algorithms in this chapter are listed below:

- *AS_i* stands for an application server, where 1≤i≤n, and n is the number of application servers we have in the system.

- *$ASPolicy_i$* is the ASPolicy assigned to *AS_i*.

- *globalCacheObjectList* is a data structure formed by merging *cacheObjectList* (described in the previous chapter) of each *AS_i*.

- *candidateObjectList* is a list of objects selected from *globalCacheObjectList* which are to be included in some *$ASPolicy_i$*.

- *candidateReplicationList* is a list of objects from *candidateObjectList* which are to be replicated at multiple sites.

- *candidateDistributionList* is a list of objects from *candidateObjectList* which are to be distributed only, i.e. they are not replicated.

- *urlList* is a list of HTTP Request URLs which are to be added to the LBPolicy.

- A HashMap urlCounter of the form URL ↔ urlASCounter[no_servers]. Here urlASCounter is an array of counters of size of the number of application servers. For each URL, urlASCounter[i] contains the number of cache objects assigned to *AS_i* by *$ASPolicy_i$* that are accessed by the HTTP request with that URL.

- *statPeriod* is the time interval of the statistics that we wish to utilise for our analysis. Thus, if we set *statPeriod to* 10 minutes, it would use the statistics of the last 10 minutes only in the analysis.

- *analysisInterval* is the time interval at which the analyser should run. Thus, at the end of each *analysisInterval*, our algorithm would run, generating the load balancing and caching strategies.

4.2 Request-unaware Algorithms

In request-unaware algorithms, we develop the caching and load balancing strategies primarily based on the statistics pertaining to cache objects. We have the following algorithm that follows this approach:

- Replication-only

4.2.1 Replication-Only

In this algorithm, we choose to have only replication of cache objects across the caches of the application servers. The ASPolicy is constructed for each application server in such a way that the most popular objects are replicated at all application servers, while there is no distribution of cache objects. Since we replicate the same objects at each site, there is no need to have a LBPolicy, the load balancer can simply use a round-robin algorithm to send the requests.

The algorithm is described below:

Input Parameters:

- REPLICATION_COUNT

Algorithm:

1. Merge *cacheObjectList* of each AS_i into a *globalCacheObjectList*. Thus, the *globalCacheObjectList* will contain all cache objects that have been accessed at any application server during *statPeriod*.

2. Sort *globalCacheObjectList* on the frequency of the *cacheObjects*.

3. Select the first REPLICATION_COUNT objects from *globalCacheObjectList* and add them to *candidateReplicationList*.

4. For each *cacheObject* in *candidateReplicationList*
    - For each $ASPolicy_i$
        - Add a mapping *cacheObjectKey* ↔ Rule (Store with Stable TTL)

5. For each AS_i
    - Send $ASPolicy_i$

4.3 Request-aware Algorithms

In request-aware algorithms, we still develop the caching and load balancing strategies based on the cache objects; however we make use of the information contained in the resource–request map of the request–resource mapping log processor to a greater extent.

We have the following three algorithms that follow this approach:

- Request-distribution
- Request-distribution with replication
- Replication-distribution

4.3.1 Request-Distribution

The request-distribution algorithm, as the name suggests, distributes the requests across application servers. In this algorithm, we do not create caching strategies for the application servers, i.e. we do not create any ASPolicy. Instead, we create a LBPolicy that distributes requests across application servers. Since a given request is sent to the same application server each time based on the LBPolicy, it is likely that the objects needed to serve that request are present in the cache of that application server. Thus, even though this is the only strategy in which we do not explicitly create a caching strategy (ASPolicy), the request distribution makes the load balancing in a way cache-aware.

In this algorithm, we determine the URLs corresponding to the most popular objects. The mapping between the URLs and the objects is obtained from the request-resource mapping log processor, and we assume that this mapping is deterministic. For dynamically generated web pages, if the objects corresponding to a URL change over time, the latest mapping would be stored at the request-resource mapping log processor. We then distribute these URLs equally amongst the application servers. Thus, for example, if we have 3000 URLs corresponding to the most popular objects, each application server would be assigned 1000 URLs.

The algorithm is described below:

Configuration Parameters:

- OBJ_FOR_REQ is the number of objects that we want to use to create the LBPolicy.

Algorithm:

1. Merge *cacheObjectList* of each AS_i into a *globalCacheObjectList*. Thus, the *globalCacheObjectList* will contain all cache objects that have been accessed at any application server during *statPeriod*.

2. Sort *globalCacheObjectList* on the frequency of the *cacheObjects*.

3. For the first OBJ_FOR_REQ *cacheObject*s in *globalCacheObjectList*
    - Determine the URLs associated with *cacheObject* (using the resource – request mapping) and add it to *urlList*

4. For each URL in *urlList*
    - Select an AS_i on a round-robin basis
        - Add a mapping URL <-> AS_i to *LBPolicy*

5. Send *LBPolicy*

4.3.2 Request-Distribution with Replication

This algorithm is very similar to the request-distribution algorithm, except that in addition to distributing the requests across the application servers, this strategy also replicates the most popular objects across the application servers. Thus, we create both caching and load balancing strategies. However it is important to note that the load balancing strategy is not derived from the caching strategy. That is, LBPolicy and ASPolicy can be created in parallel.

The algorithm for this strategy is described below:

Input Parameters:

- REPLICATION_COUNT

Configuration Parameters:

- OBJ_FOR_REQ is the number of objects that we want to use to create the LBPolicy.

Algorithm:

1. Merge *cacheObjectList* of each AS_i into a *globalCacheObjectList*. Thus, the *globalCacheObjectList* will contain all objects that have been accessed at any application server during *statPeriod*.

2. Sort *globalCacheObjectList* on the frequency of the *cacheObjects*.

3. For the first OBJ_FOR_REQ *cacheObject*s in *globalCacheObjectList*
    - Determine the URLs associated with *cacheObject* (using the Resource – Request Mapping) and add it to *urlList*

4. For each URL in *urlList*
    - Select an AS_i on a round-robin basis
        - Add a mapping URL <-> AS_i to *LBPolicy*

5. Select the first REPLICATION_COUNT objects from *globalCacheObjectList* and add them to *candidateReplicationList*.

6. For each *cacheObject* in *candidateReplicationList*
    - For each $ASPolicy_i$
        - Add a mapping *cacheObjectKey* ↔ Rule (Store with Stable TTL)

7. For each AS_i
    - Send $ASPolicy_i$

8. Send *LBPolicy*

4.3.3 Replication-Distribution

In this algorithm, we choose to have a mix of replication and distribution of cache objects across the caches of the application servers. The ASPolicy is constructed for each application server in such a way that the most popular objects are replicated on all application servers, while a fraction of the rest are distributed across them. We can specify a threshold to determine which objects are to be classified as “most popular”. In order to create the LBPolicy, we examine the ASPolicy for each application server and make a list of URLs accessing the objects in each ASPolicy. Then, we assign the URL to the application server which has been assigned the maximum number of objects accessed by that URL.

The algorithm for this strategy is described below:

Input Parameters:

- REPLICATION_COUNT
- DISTRIBUTION_COUNT

Algorithm:

1. Merge *cacheObjectList* of each AS_i into a *globalCacheObjectList*. Thus, the *globalCacheObjectList* will contain all objects that have been accessed at any application server during *statPeriod*.

2. Sort *globalCacheObjectList* on the access frequency of the *cacheObjects*.

3. Select the first REPLICATION_COUNT objects from *globalCacheObjectList* and add them to *candidateReplicationList*.

4. Select the next DISTRIBUTED_COUNT objects from *globalCacheObjectList* and add them to *candidateDistributionList*

5. For each *cacheObject* in *candidateReplicationList*
    - For each $ASPolicy_i$

        - Add a mapping *cacheObjectKey* ↔ Rule (Store with Stable TTL)

6. For each *cacheObject* in *candidateDistributionList*
    - Select an AS_i on a round-robin basis and for $ASPolicy_i$
        - Add a mapping *cacheObjectKey* ↔ Rule (Store with Long TTL)

7. For each $\mathrm{ASPolicy}_\mathrm{i}$
    - For each $cacheObjectKey_i$ in $ASPolicy_i$
        - For each URL that accesses the cache object with key *cacheObjectKey* (we can determine the URLs using the request – resource mapping log processor)
            - Retrieve the mapping corresponding to URL from *urlCounter* (Create a new one if it does not exist)
            - *urlASCounter*[i]++

8. For each URL in *urlCounter*
    - Determine the maximum value amongst *urlASCounter*[i]
    - Add a mapping URL ↔ IP Address of AS_i to *LBPolicy*

9. For each AS_i
    - Send $ASPolicy_i$

10. Send *LBPolicy*

4.4 Summary

Thus, we have seen the different algorithms that have various levels of complexities. In Table-1 we can see how these algorithms can be categorised based on what type of strategies they generate (load balancing/caching):

Algorithm	**LBPolicy**	**ASPolicy**
Replication-only	No	Yes
Request-distribution	Yes	No
Request-distribution with replication	Yes	Yes
Replication – distribution	Yes	Yes

Table-1 Categorizing algorithms based on strategies produced

We can see from the table that the replication-only and the request-distribution algorithms are the most simple algorithms which produce only caching and load balancing strategies. The other two algorithms are more complex in nature, producing both caching and load balancing strategies.

Chapter 5

Experimental Results and Evaluation

5.1 Introduction

In this chapter, we first explain certain specific implementation details regarding the monitoring tool. Then, the software and hardware environment in which the experiments have been carried out is described. We also explain details about the benchmark RUBiS that we use for evaluating our algorithms. After that experimental results are shown. We analyse different scenarios and cases, followed by a discussion on the suitability of the developed algorithms.

5.2 Implementation Details

In this section, we discuss some implementation details regarding the logging that we do at the application servers, the GUID that we discussed earlier in Section 3.1.2 and some changes that we have made to Hibernate for our implementation.

5.2.1 Logging

Logging is done in the log capture component at the application server using the open source log4j [22] logging API for Java. Log4j has three main components: loggers, appenders and layouts. These three types of components work together to enable developers to log messages according to message type and level, and to control at runtime how these messages are formatted and where they are reported. Loggers are the main component of log4j. Loggers help to define a hierarchy for the logging and allow us to decide at run-time which log statements to print or not. A log statement is printed depending on its level and its logger.

Log4j allows logging requests to print to multiple destinations. In log4j terminology, an output destination is called an appender. It is possible to have the output destination to be a variety of components such as console, files, GUI and remote socket servers. The remote socket server appender is known as *SocketAppender*. SocketAppender sends the logging events directly to a remote location as described in Section 3.2.1.

5.2.2 GUID

The need for a Globally Unique Identifier (GUID) is mentioned in Section 3.1. The architecture of JBoss is such that when a request arrives at the application server, JBoss gets a thread from a thread pool, and subsequently the entire request is processed in the same thread. By storing the GUID in a thread-local variable in Java (a `ThreadLocal` class object), we can insert the same GUID in both the access logs and the cache logs, and subsequently map HTTP requests with the objects in the log processors. The GUID is a 32 byte string which is encoded to 24 bytes using a fast base64 encoder. We make use of the Mapped Diagnostic Context (MDC) of log4j. The

MDC is a map which stores the context data of the particular thread where the context is running. Each thread has its own MDC which is global to the entire thread. Hence, we put the GUID inside the MDC, which is later extracted inside the log processors. We also need to store the hostname of the application server inside the MDC, so that the log processors get to know the source of the log.

5.2.3 Hibernate

As described earlier, we use two types of logs, access logs and cache logs. The access log is the log generated at the Interceptor. The cache logs that log the fetch and the put calls in the second-level cache of Hibernate are generated by actually adding the log statements inside the Hibernate code. For example, in the code, just before the statement where a call to fetch the object from the second-level cache is made, we log the event and it is sent to the log processing and analysis component.

5.3 Hardware Platform

All machines, which include the client emulator, load balancer, application servers and the database, have the same hardware configuration. Each machine has an Intel (R) Core (TM) 2 CPU 2.66GHz processor with 8 GB RAM. The machines share a file system which has a 542 GB hard drive. All the machines are connected through a 1 Gbps LAN switch.

5.4 Software Platform and Architecture

All the experiments performed are on a multi-tier architecture. The client machine that sends the requests interacts with the load balancer. The load balancer is placed in front of a cluster of application servers that act as the application tier and a database server which is a part of the database tier. All the machines on the cluster lie on a single LAN. All the experiments except those discussed in Section 5.6.2.7 use 3 application servers.

All the machines run on *Fedora 13 (Goddard)* [23] with Linux kernel version *2.6.27.25-78*. We use *JBoss 5.1.0* [24] as the Java EE compliant application server that is installed on the machines at the application tier. *Hibernate* [3] is used for providing ORM (Object Relational Mapping) at the JBoss AS. We use Ehcache 2.0.1 [29] as the second-level cache provider in Hibernate. The Ehcache source code has been modified to incorporate the functionality of the distributed co-operative cache described in Section 2.5, and also to support certain extensions such as enabling replication of objects in the cache which have been developed by us. For persistence, we use *PostgreSQL 8.4.1* [25] as our database server. All the machines have Java from Oracle with version 1.6.0_25 installed.

For load balancing, we have *Apache HTTP Server Version 2.2* [26] installed. It acts as a dispatch-level load balancer. Apache comes with a set of core modules and a set of other modules which can be installed if required. We utilise the *mod_proxy* and the *mod_proxy_balancer* [27, 28] modules of Apache HTTP server for load balancing (which are not present in the default installation). In our setup, the Apache HTTP server works as a reverse proxy, where the server acts as a single gateway to all the application servers that are behind the load balancer. It distributes requests to the application servers as per the load balancing algorithm which is in force at the load balancer, and also sends the response back to the client from the application server in a manner which is completely transparent.

In order to implement the load balancing strategy using the LBPolicy created by the analyser, we have made changes to the mod_proxy_balancer module of the Apache server, so that once the strategy is received at the load balancer, that strategy is used for load balancing instead of the default algorithm. We choose to run the *Request Counting* algorithm as the default algorithm, which already comes with the Apache installation. The request counting algorithm allows us to configure each member of the cluster behind the load balancer to receive their fair share of work based on the total number of requests a particular member should handle. Suppose that we indicate a share of a member's work by *lbfactor*, and we have the configuration as shown in Table-2.

	Server 1	Server 2	Server 3
lbfactor	1	2	1

Table-2 Sample configuration for request counting algorithm

In this case, Server 2 would be handling twice the number of requests as Server 1 and Server 3 do. The algorithm keeps track of the number of requests that have been directed to each server, and accordingly it sends the next request to a server such that the ratio of requests served across the servers remains as configured above. Thus, if the lbfactor for all servers is kept the same, it turns out that the request counting algorithm simplifies into a round robin algorithm.

5.5 Benchmark

We use the RUBiS benchmark [29] in our experiments to evaluate our algorithms. It is a free and open-source auction site prototype which is modelled after the highly popular action site eBay.com. It is chiefly used to evaluate performance and scalability of application servers and also to evaluate various application design models. RUBiS implements the most common type of functionality that an auction site offers, namely selling, browsing and bidding. For our evaluation purposes, we only use the browsing mix since it consists of read-only transactions. Our algorithms develop caching and load balancing strategies, and with a read-only workload we can best utilise the cache to our advantage. In contrast, write transactions cause updates to the database, putting greater load on the database server. Furthermore, we would need a cache invalidation mechanism if we decided to use write-transactions as a part of the workload, which was out of the scope of this thesis. Moreover, we would also have to take into account consistency issues of the data cached across the servers if we were to use a workload that contained any write operations. Thus, for reasons of simplicity we used only the read-only workload.

A client-browser emulator (written in Java) is also available along with the application. For each customer session, the emulator starts a persistent HTTP connection to the web server (in our case the load balancer), and closes the connection at the end of the session. A session consists of a sequence of interactions (having different transactions) for the same customer. There is also a think time for each customer before the next interaction starts. The think time and session time are generated from a negative exponential distribution with a mean of 7 seconds and 15 minutes respectively. The number of concurrent clients determines the load submitted to the system. As the number of clients increase, the load on the system increases. At the end of each run, the client produces performance results of the run with useful statistics such as average response time, throughput, network traffic, system statistics etc. The characteristics of each run such as workload, runtime, warm-up period, server information, etc. are set through a properties file.

There are about 16 different interactions that can be performed from the client's browser such as browsing items by category/region, bidding, buying/selling of items, viewing user profiles or comments on the items. The sequence of interactions depends upon a state transition matrix that defines the probability to go from one interaction to another one. One can change this transition matrix according to the workload mix that we wish to follow. In order to vary the load on the site, we can adjust the number of clients that run concurrently.

The RUBiS application itself has to be deployed on an application server (in our case it is JBoss). The application has both static and dynamic content. The static content consists of the HTML pages and images. The dynamic content deals with the database. Several versions of RUBiS are available which have been implemented using PHP, Java Servlets and EJB (Enterprise Java Beans). For our experiments, we use the Java Servlets version.

5.6 Experiments and Results

In this section, we first discuss the configuration details for the experiments that we run, followed by an evaluation of the performance of the various algorithms that have been described in the previous chapter.

5.6.1 Configuration

As mentioned in the previous section, we use only a read-only workload for our experiments. So, the transactions in RUBiS follow a browsing mix workload. The transition matrix is set so that the following pages are hit:

1. Home
2. Browse
3. BrowseCategories
4. SearchItemsInCategories

5. BrowseRegions
6. BrowsCategoriesInRegions
7. SearchItemsInRegions
8. ViewItem
9. ViewUserInfo
10. ViewBidHistory

Out of these pages, Home and Browse are static pages, while the rest are dynamic, i.e. their content is fetched from the database.

The RUBiS database size that we use in our experiments is 1 GB. The size of the cache at each application server is set to 30,000 objects (approximately 60 MB) in the experiments unless specified otherwise. We run each of our experiments for 1 hour. This consists of 30% warm-up time and a 15% down-ramp time whose results we do not include in the performance figures. All experiments are performed with 3 application servers unless specified otherwise. In most of our experiments, we try to show our results at two different scenarios: when the system is running at low loads, and the other when the system is nearly saturated.

5.6.2 Results

5.6.2.1 Access Times

In order to better understand our results and the reasons for them, we first present a rough approximation of the time it takes to fetch a single object from the cache of the application server locally (a local cache hit), from the cache of another application server (a remote cache hit) and the time it takes to fetch an object from the database. Note that the figures are only an approximation, which are calculated at a time when there is no load on the system. It is difficult to get an estimation of the same when the experiment is running, and the figures would greatly vary in the case of fetching the data from the database especially with different types of queries, database sizes and object sizes. Also, when there is a cache hit (local or remote), the objects are fetched one-by-one, whereas when there is a need to fetch the objects from the database, the objects may be returned together as part of a single query. Further, there is some overhead involved of the application itself and other processing which is difficult to separate. Hence, an accurate comparison is difficult to establish.

In Table-3, we can see the difference between the access times. The access time in case of a local cache hit is the lowest, whilst the time for a remote hit is slightly better than accessing an object from a database. The database tier is often the bottleneck in a multi-tier system and this figure is expected to go up significantly in a high workload situation, whilst the access times for local and remote cache hits are not expected to shoot up too much unless the application server gets overloaded. We calculate these times as follows by modifying the Ehcache source code:

- **Local cache hit**: Measuring the system time before and after the *get* call to the local (second-level) cache

- **Remote cache hit**: Measuring the system time before and after the remote call to another application to fetch the object from that server's cache

- **Database hit**: Measuring the system time at the end of the *get* call (which returns *null* since it is neither present in the local or remote cache) and the beginning of the *put* call to put the object in the local cache (since the object fetched from the database is put in the local cache immediately) in the Ehcache source code that we have modified

Type of Access	Approximate Time (in ms)
Local cache hit	0.2
Remote cache hit	2
Database hit	2.2

Table-3: Access times for local cache hit, remote cache hit and database hit

Thus, the maximum gain which can be obtained from our strategies is when there is a *local cache hit*. Our strategies try to maximize the number of local cache hits in order to improve performance.

5.6.2.2 Ehcache v/s Distributed Co-operative Cache

We use the distributed co-operative cache on top of the default Ehcache since the co-operative cache has significantly less response time and higher throughput. The co-operative cache can store more objects than Ehcache as the caches co-operate and hence have a combined cache capacity greater than in the case of Ehcache where the cache at each application server is independent of the other caches. Having a greater cache size means that there are less frequent accesses to the database, and so the database server does not saturate as soon as in the case of Ehcache. Although there is some additional overhead related to maintaining this distributed cache, it still outperforms the Ehcache.

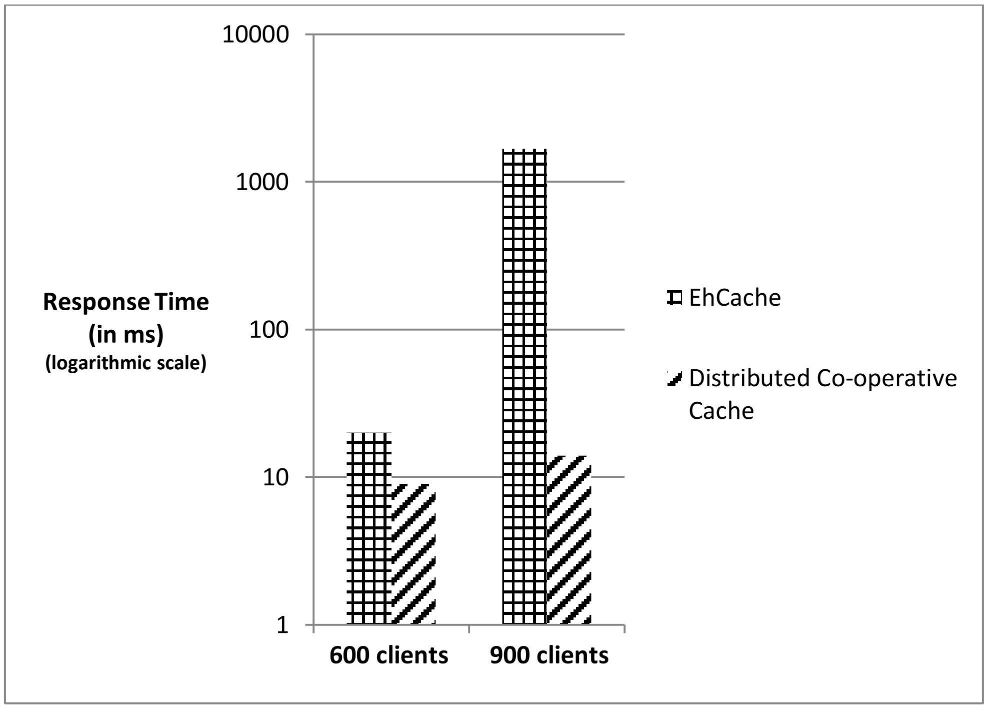

Figure 5-1 Comparison of response times for Ehcache and distributed co-operative cache at 600 and 900 clients

We can see in Figure 5-1 that 600 clients, when there is not much load on the system, the distributed co-operative cache performs better by about 35%. At 900 clients, Ehcache has already reached saturation with an average response time of 1683 ms, whilst the distributed co-operative cache is still performing well. Thus, the distributed co-operative cache has better scalability than Ehcache.

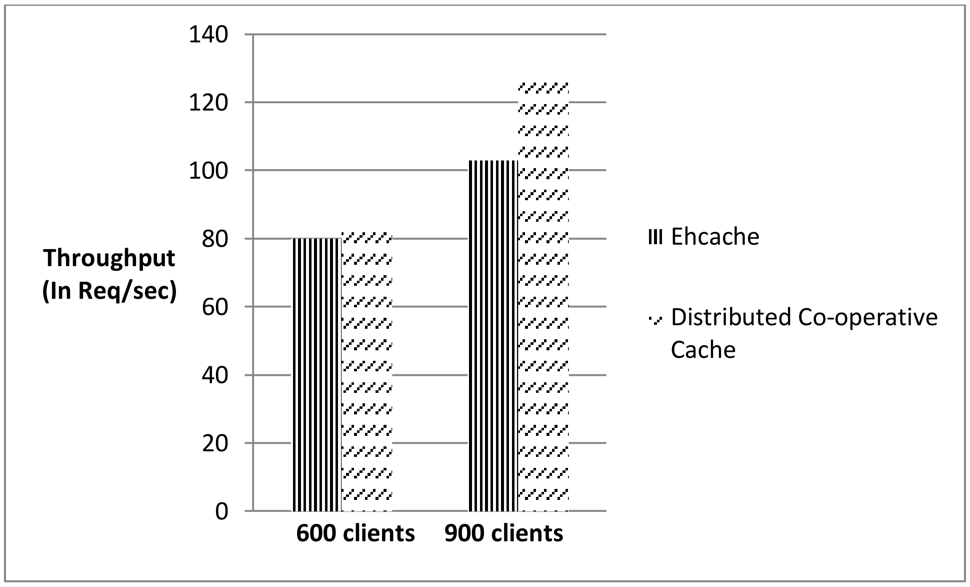

Figure 5-2 Comparison of throughput for Ehcache and distributed co-operative cache at 600 and 900 clients

In Figure 5-2, we can see the throughput (measured by number of requests per second served) for Ehcache and distributed co-operative cache. At 600 clients, we see a 6% increase in throughput in the distributed co-operative cache as compared to Ehcache. We subsequently observe a marked increase in throughput from 103 req/sec to 128 req/sec (about 25%) from Ehcache to the distributed co-operative cache at 900 clients. As Ehcache is saturated at 900 clients, it can no longer serve the full load.

The reason for this improvement in response time and throughput is mainly due to the increased second-level cache hit rate. In Table-4, we see comparison of the hit and miss count for both cases. For the distributed co-operative cache, a hit at the local cache or a remote cache is a cache hit.

	Second – Level Cache Hit Count	**Second – Level Cache Miss Count**
Ehcache	683483	176244
Distributed Co-operative Cache	1407207	22950

Table-4 Comparison of second-level cache hit / miss counts for Ehcache and distributed co-operative cache

5.6.2.3 Logging Overhead

Since only the log capture component of the monitoring tool is present at the application server, the monitoring tool itself does not have a lot of performance overhead. The log processing and analysis component, which does the bulk of the processing done by the monitoring tool, is located on a separate server, and hence it does not affect the performance of the application itself. In Figure 5-3, we can see the average response times for the distributed co-operative cache when the logging in the monitoring tool is enabled, and when it is disabled. For a normal workload at 900 clients, and even at 1800 clients when the database server has started to saturate, we see an approximately 1% degradation in performance, which is negligible.

Thus, we can see that the logging itself has minimal impact on the performance; however, we need to run a separate server for the log processing and analysis component for this purpose.

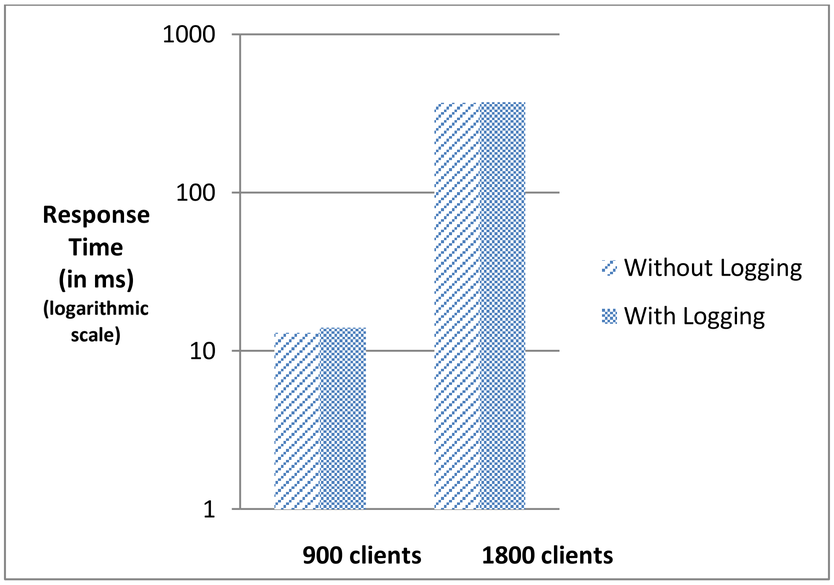

Figure 5-3 Comparison of response times for distributed co-operative cache at 900 and 1800 clients with and without logging

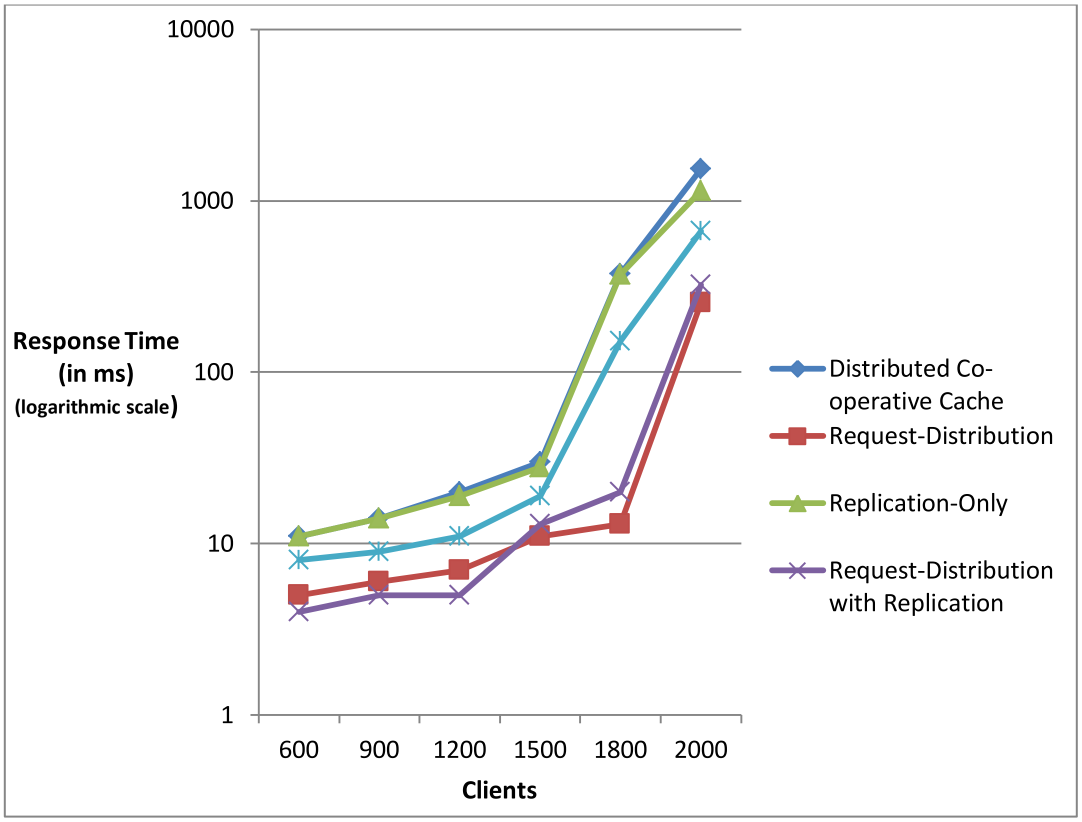

Figure 5-4 Comparison of response times for various strategies

5.6.2.4 Comparative Evaluation of Algorithms

Next, we discuss the performance of our algorithms. We compare their performance against the distributed co-operative cache since they are built on top of this infrastructure. In Figure 5-4, we can see average response times with increase in number of clients for the distributed co-operative cache, request-distribution, request-distribution with replication, replication-only and the replication-distribution algorithms.

For the request-distribution algorithm, we take the 4000 most popular objects (as determined at the log processing and analysis component), determine the URLs corresponding to them and distribute them equally across the servers. For request-distribution with replication, we take the 4000 most popular objects, determine the URLs corresponding to them and distribute them equally across the servers. In addition to that we also replicate these 4000 objects at each server. For replication-only, we take the 4000 most popular objects and replicate these objects in the cache of each of the servers. For the replication-distribution, we take the 4000 most popular objects and replicate these objects in the cache of each of the servers. In addition to that, we distribute the rest of the objects equally amongst the servers.

For 600 clients, the request-distribution algorithm shows an improvement of about 50% as compared to the distributed co-operative cache. In addition to that it reaches saturation a lot later than the distributed co-operative cache. At 2000 clients, the distributed co-operative cache has already reached saturation and the response time has reached 1525 ms, while for the request-distribution algorithm it is 255 ms. The request-distribution with replication algorithm performs the best at lower loads (until 1200 clients), but after that the request-distribution algorithm outperforms it. We see that the replication-only is slightly better than the distributed co-operative cache, however its curve follows the distributed co-operative cache curve more closely, and performs better than the distributed co-operative cache at higher loads. The replication-distribution algorithm performs better than the replication-only algorithm, however still it does not match the performance achieved by the two request-distribution algorithms.

Thus, to conclude, we see that the replication-only algorithm does not offer much improvement compared to the distributed co-operative cache. The request-distribution and request-distribution with replication algorithms have the best performance of all the algorithms. The replication-distribution algorithm, although being the most sophisticated one on paper, is better than the replication-only algorithm but does not match the performance of the request distribution and request distribution with replication algorithms.

The chief contributing factor behind the behaviour of these algorithms is the access times that have been shown in Table-1. In the request-distribution algorithm, we develop a load balancing strategy that consists of distributing the requests associated with the most popular objects across the servers. This leads to the most popular requests being sent to the same server each time, and thus, there is a greater probability that all the objects that are required to serve this request are present in the local cache only, whose access time is the lowest. For the request-distribution with replication algorithm, we see that for lower workloads, it performs slightly better than the strategy without replication. This can be explained by the fact that since the most popular objects have been replicated at all sites; there are no remote calls for these objects, and thus, they can be accessed from the cache locally which reduces response time. As the load increases with increasing number of clients, the number of objects accessed increases, and hence replication decreases the combined cache capacity. In this case, replication of objects is no more advantageous.

For the replication-only algorithm, we do not have a load balancing strategy and there is a caching strategy sent to the application servers that the 4000 most popular objects are replicated at each site. Thus, an individual request type is not always sent to the same server, and hence there is a much lower probability of all of the objects corresponding to the request being present at that particular site. This results into greater number of remote accesses (where an object is fetched from the cache of another application server). Since a remote access takes more time than a local access, the replication-only algorithm does not perform as well as the request-distribution algorithm. The replication-distribution algorithm performs better than the replication-only algorithm since it produces a load balancing strategy as well which the latter does not. In this algorithm, a URL is assigned to the server which is assigned the maximum number of objects in the caching strategy which has been developed by it. This leads to a decrease in the number of remote hits compared to the replication-only algorithm. However, since we distribute the non-replicated objects equally but randomly amongst the remaining servers, it does not perform as well as the two request-distribution algorithms, which have more local hits.

In Figure 5-5, we show a comparison of the number of local hits and remote hits.

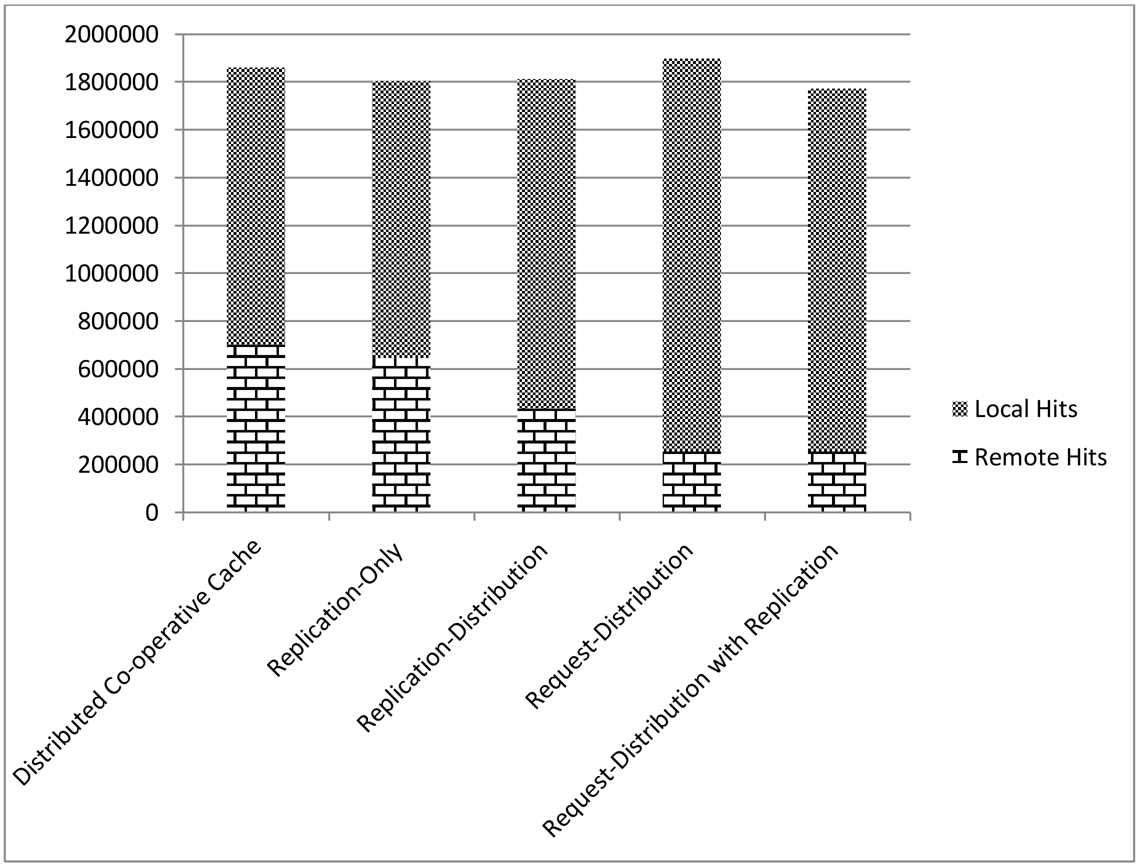

Figure 5-5 Comparison of local and remote hits at 2000 clients

5.6.2.5 Parameter Variation

We also try to give an insight how various parameters that have been used in developing our strategies may impact performance. As an example, we take the replication-only algorithm. We change the number of objects to be replicated at each site from 4000 to 8000 and see how this influences our results. In Table-5, we see the results of changing this parameter. When we increase the number of objects to be replicated to 8000, we see a slight increase in response time, and decrease in the throughput achieved. The reason for this is since we replicate 8000 objects at each server, it takes up space equivalent to that of 24,000 objects across the caches at all servers. While for 4000 objects it would take up 12,000 objects. Having a higher number of objects replicated at each server leads to lesser cache size available for the remaining objects. Hence, although having more objects replicated across all the servers may lead to lesser remote hits, it effectively decreases overall cache capacity. As a result, as the number of objects in the cache exceeds its capacity, replication tends to degrade performance rather than improve it. Thus, it is important to select the correct parameters while implementing a particular algorithm for optimal performance.

	Response Time (in ms)	**Throughput (in req/sec)**
Replication-only - 4000 Objects	19	171
Replication-only - 8000 Objects	23	168
Distributed co-operative cache	20	171

Table-5 Varying strategy parameters for 1200 clients

5.6.2.6 Replication

In this section, we have a closer look at replication of objects in the cache. From Section 5.6.2.3 and 5.6.2.4, we gather that replication needs to be handled very carefully, else it can degrade performance. In Section 5.6.2.3, we see that at higher workloads, the request-distribution algorithm performs better than the request-distribution with replication algorithm, and in Section 5.6.2.4 we observe that increasing the number of objects to be replicated beyond a certain limit has a detrimental effect on performance. Here, we try to demonstrate that in certain scenarios replication is beneficial and improves performance. This is particularly true for applications where a very high percentage of requests access a small subset of data items. Thus, these data items are very frequently accessed.

In order to create such a workload wherein there is a small set of data that is extremely frequently accessed, we change the state transition matrix of RUBiS. There are only 62 distinct *Regions* and 20 distinct *Categories* in the RUBiS database. We change the share of the BrowseCategories and BrowseRegions pages to about 35% from the 2% that we use for the rest of our experiments to show the significance of replication.

Figure 5-6 shows the response time of the distributed co-operative cache, request-distribution and request-distribution with replication algorithms. Since the total number of objects accessed for this workload is significantly smaller than for the other experiments, we see that even at 2000 clients, the database server does not saturate. We see that the distributed co-operative cache has a response time of 14 ms, and the request-distribution algorithm has an approximately 21% improvement over the distributed co-operative cache. However, in this case we see that even at a higher workload, the request-distribution with replication algorithm shows about 18% decrease in response time.

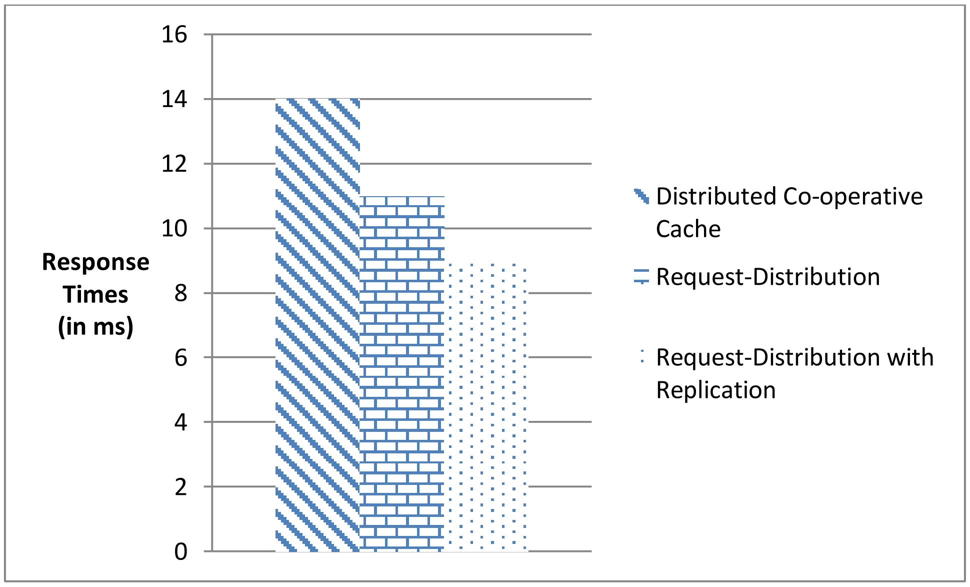

Figure 5-6 Response times for 2000 clients for a workload that accesses a small but highly popular set of objects

Thus, we can conclude that replication is an important tool to improve performance depending on the workload characteristic of the application we are using.

5.6.2.7 Cache Size

Next, we examine how the difference in cache size affects performance. We ran our experiments at a cache size of 110 MB as compared to the 60 MB cache size used in the previous experiments, and see the difference in our results. Figure 5-7 shows the difference in response times for the distributed co-operative cache and our replication-only algorithm at 1200 and 2000 clients with a cache size of 60 MB and 110 MB. At 1200 clients, the system is running at low loads, while at 2000 clients, the load is enough to nearly saturate the system. For the replication-only algorithm, we replicate the 4000 most popular objects at each site.

We see that for distributed co-operative cache there is a 30% improvement in response time when there is an increase in the cache from 60 MB to 110 MB at 1200 clients. The replication-only algorithm also shows an improvement of 37% in response time for the same case. For 2000 clients, we see that by increasing the cache size, there is less saturation of the database for both the distributed co-operative cache and replication-only algorithm. As the cache size increases, the number of local hits increases as well as the total combined capacity of the cache increases, thus resulting into less frequent calls to the database. This leads to less saturation at the database server and thus, we can see an improvement in performance when the cache size is increased. Since we are using a read-only workload, there is not much additional overhead involved in increasing the cache size.

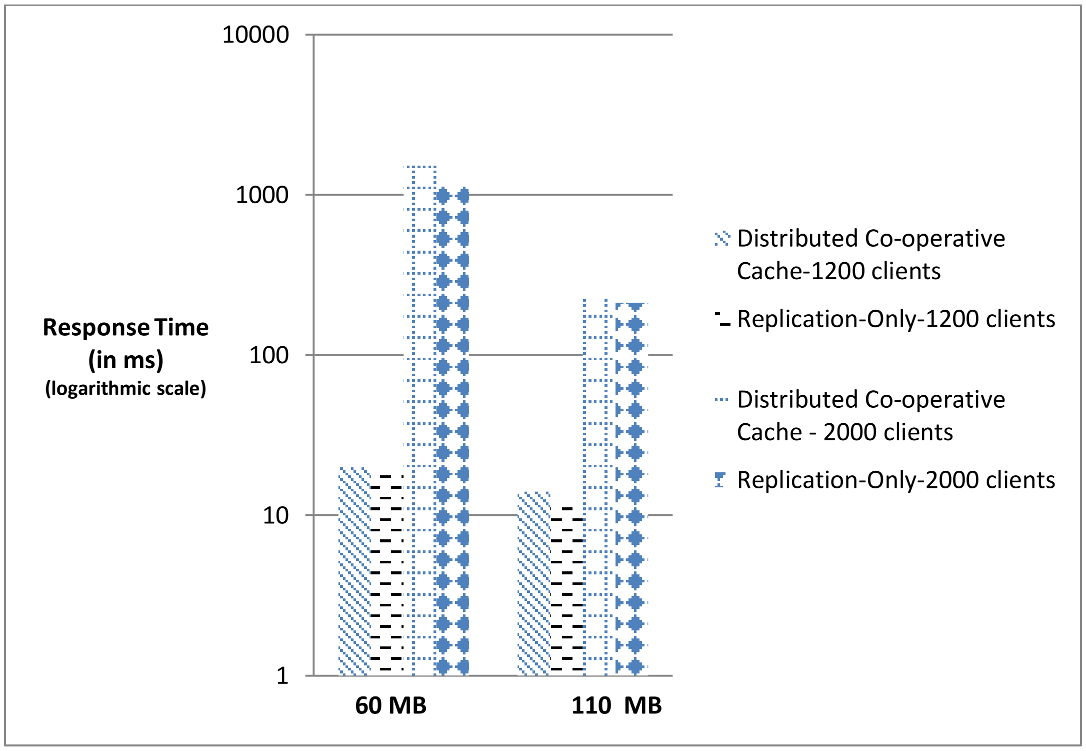

Figure 5-7 Response times at 60 MB and 110 MB cache for 1200 and 2000 clients

5.6.2.8 Scalability

Finally, we have a look at the scalability of our solutions in terms of number of application servers. All our previous experiments have been run on an application tier that consisted of 3 servers. We increase the number of servers to 5, and see what effect it has on performance.

Figure 5-8 shows the difference in response times for both the distributed co-operative cache and the request-distribution algorithm at 1800 and 2000 clients. We can see a significant improvement in response time for the distributed co-operative cache, from 375 ms for 3 servers to 13 ms for 5 servers at 1800 clients. This is mainly because for the 3 server case, the database server has started to saturate. For our request-distribution algorithm we see a 50% decrease in response time for 5 servers as compared to 3 servers. A drop in response time similar to that of the distributed co-operative cache is also seen for the request-distribution algorithm when we utilise 5 servers.

Although there might be some additional overhead associated with the system in case of 5 servers, the overall cache capacity is increased in the case of 5 servers, and hence the number of local hits and remote hits will also be more, resulting in fewer number of database accesses. It is important to note that the RUBiS application that we are using does not have any CPU-intensive operations; hence the gain achieved by increasing the number of servers is primarily due to an increased cache capacity and not increased processing capacity.

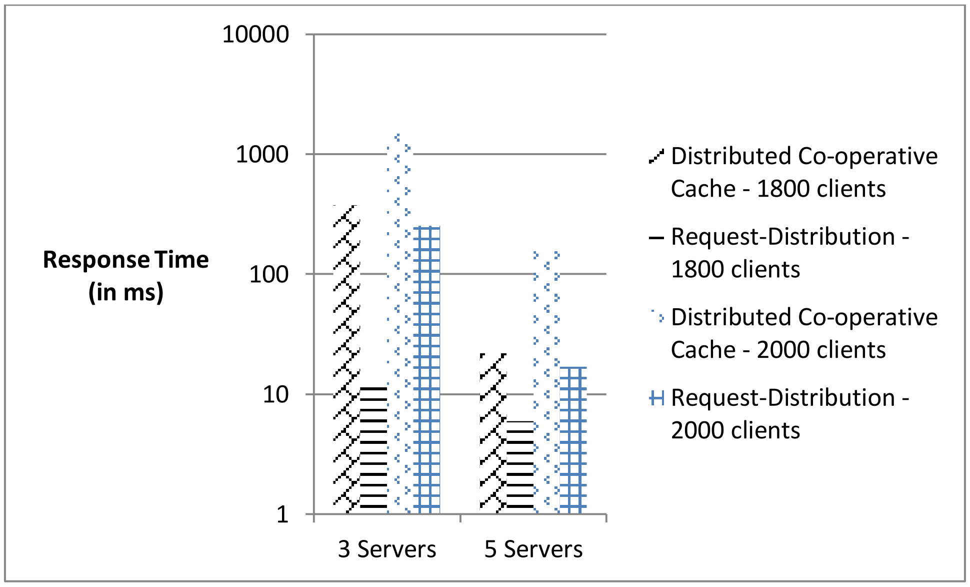

Figure 5-8 Response times at 1800 and 2000 clients for 3 servers and 5 Servers

We also see that having 5 servers not only reduces response time by increasing the overall cache capacity, but it also increases throughput that can be achieved. We see the difference in throughput in Figure 5-9. There is a marked increase of about 5% in throughput for the distributed co-operative cache that has reached saturation in the case of 3 servers. For the request-distribution algorithm too, we see that the throughput increases about 1% from 254 req/sec to 257 req/sec. Thus, this shows that our strategies are indeed *scalable.*

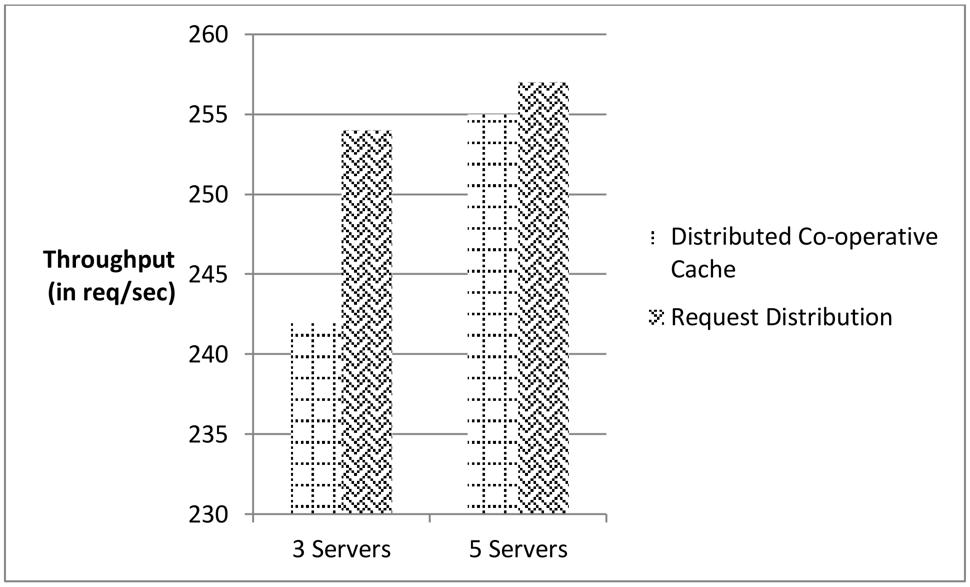

Figure 5-9 Throughput at 1800 clients for 3 servers and 5 Servers

5.7 Summary

We presented several different experiments and scenarios to evaluate the performance of our algorithms. Although our algorithms are application-independent, it is likely that the best algorithm in a given situation depends on various factors such as type of application, workload and memory constraints.

However, we see a marked trend in all the experiments that having a load balancing strategy is beneficial regardless of other conditions. This is because having a load balancing strategy leads to a higher probability of a request being served with maximum local cache hits. We also see how replication affects performance, both at low and higher loads. If utilised in an optimal fashion, it can aid in enhancing performance.

Chapter 6

Conclusions and Future Work

In this chapter, we summarize our findings and discuss the main contributions made by this thesis. We also have a look at the possible directions in which this thesis can be extended.

6.1 Conclusions

As a first contribution, we have presented a monitoring tool that has been developed for the purpose of creating the necessary infrastructure for the generation of our caching and load balancing strategies. The monitoring tool itself is application-independent and thus, can be used across applications. This is a major advantage, since as a result of that our solution is not tightly coupled with any specific application. Moreover, since the monitoring tool does not consume much processing time at the application server itself, and instead does the bulk of processing at a separate server, it has minimal effect on the actual application running at the application servers.

The algorithms developed in this thesis utilize the information about both the requests and the objects they access. We have run experiments that evaluate the performance of our caching and load balancing strategies. The foundation of our strategies is based on our observation that increasing the number of local cache hits leads to a notable improvement in performance. It is clear that an effective load balancing strategy is a critical component if we are to achieve scalability and significant improvement in performance in terms of response time and throughput achieved. The other vital aspect about developing effective caching strategies is replication. We have seen in several of our experiments that replication can be useful and aid in improving performance if utilized well, but if not handled well, we can see that it can actually have a detrimental effect on performance.

In the experiments mentioned in Section 5.6.2.7, we saw that an increase in the number of application servers resulted into better performance, demonstrating that our strategies are scalable. With the exponential growth seen in e-commerce sites and other internet-based services and applications, it is crucial that any solution proposed is scalable; otherwise it would have no practical value. It is important to note that our algorithms have been primarily targeted towards read-only transactions; we would need to incorporate a cache consistency mechanism in our algorithms for an environment with write transactions as well. With increase in the number of application servers, the cache consistency mechanism would get increasingly complex, possibly affecting the scalability of our algorithms.

To summarize, our experiments indicate that load balancing algorithms that are cache-content aware can result into significant performance gains.

6.2 Future Work

There are a few directions in which the work in this thesis can be extended upon.

Firstly, we work with a read-only cache which does not have any write operations to the database. It would be interesting to see how our strategies fare with a write-intensive workload. However, for that we would need to incorporate a cache-invalidation mechanism that maintains the consistency in the caches, i.e. it ensures that the data in the cache is consistent with the data in the database, and it is also consistent across all the caches.

Another possible direction is to explore different methods of analyzing the logs at the log processing and analysis component. At present, the analysis uses the frequency of objects as the main parameters while developing the caching and load balancing strategies. A more complex mechanism may be used for this purpose, such as machine learning techniques. For example, we could automate the process to select the right parameters to be used to develop the strategy such as number of objects to be replicated. One could also look at other parameters such as CPU utilization, memory usage, etc. at the application servers while developing the strategy.

Currently we perform all the analysis on a separate machine. It might be possible to put some intelligence at the load balancer itself, thereby reducing the workload on the separate server. In this thesis, for the workloads and the benchmark we are using, there is not much load at the machine which hosts the log processing and analysis component, but that might not be the case with other applications.

Bibliography

[1] Cardellini, V., Colajanni M., and Yu, P.S. *Dynamic load balancing on web-server systems.* IEEE Internet Computing, 1999. **3**(3):p. 28-39.

[2] JBoss AS 4.0 Guide – Jboss Community. [Online] http://docs.jboss.org/jbossas/jboss4guide/r1/html/pr05.html#hibernate

[3] Hibernate Reference Documentation 3.6.3. [Online] http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/performance.html

[4] Ali, Shamir. *Contquer: An optimized distributed cooperative query caching architecture.* MSc Thesis. McGill University, Montreal, Quebec, Canada, 2011.

[5] Ehcache. [Online] http://www.ehcache.org/.

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

[7] Tickoo, Neeraj. *Cache aware load balancing for scaling of multi-tier architectures.* MSc Thesis. McGill University, Montreal, Quebec, Canada, 2011.

[8] 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. 8-13.

[9] Zhang, X., Barrientos, M., Chen, J., Seltzer, M. *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.

[10] Cherkasova, L. and Karlsson, M. *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.

[11] Sivasubramanian, S., Pierre, G. and Steen, M. *A case for dynamic selection of replication and caching strategies*, in *Proceedings of the Eighth International Workshop on Web Content Caching and Distribution, September-October 2003*.

[12] Pierre, G., Kuz, I., Steen, M. and Tanenbaum, A. *Differentiated strategies for replicating web documents* in *Computer Communications.* (Elsevier Editor), 24 (2) (2001) pp. 232-240.

[13] Kemme, B., Patiño-Martínez, M., Jiménez-Peris, R., Salas, J. *Exactly-once interaction in a multi-tier architecture*. In *VLDB Workshop on Design, Implementation, and Deployment of Database Replication*, August 2005.

[14] Perez-Sorrosal, F., Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B. *Consistent and scalable cache replication for multi-tier J2EE applications*. In *ACM/IFIP/USENIX Int. Middleware Conf.*, Nov. 2007.

[15] Attar, M., Ozsu, M.T. *Alternative architectures and protocols for providing strong consistency in dynamic web applications*. WWW Journal (2006).

[16] Frølund, S., Guerraoui, R. *e-transactions: End-to-end reliability for three-tier architectures*. In *IEEE Trans. Software Engineering*. 8(4) (2002) pg 378–395.

[17] Amza, C., Soundararajan, G., and E. Cecchet. *Transparent caching with strong consistency in dynamic content web sites*. In *International Conference on Supercomputing* (ICS), pages 264–273, 2005.

[18] Lillis, K. and Pitoura, E. *Cooperative XPath caching*. In *ACM SIGMOD Conf*., 2008.

[19] Cohen, E. and Shenker, S. *Replication strategies in unstructured peer-to-peer networks*. In *ACM SIGMOD Conference*, pages 177–190, 2002.

[20] Skobeltsyn, G. and Aberer, K. *Distributed cache table: efficient query-driven processing of multi-term queries in P2P Networks*. In *Workshop on Information Retrieval in Peer-to-Peer Networks* (*P2PIR)*, 2006.

[21] Iyer, S., Rowstron, A. and Druschel, P. *Squirrel: A decentralized peer-to-peer web cache*. In *Principles of Distributed Computing (PODC),* 2002.

[22] Apache Logging Services, log4j 1.2. http://logging.apache.org/log4j/1.2/

[23] Fedora Project Releases. [Online] http://fedoraproject.org/wiki/Releases

[24] JBoss AS 5 Documentation. [Online] http://www.jboss.org/jbossas/docs/5-x

[25] PostgreSQL 8.4.1 Documentation. [Online]
http://www.postgresql.org/docs/8.4/static/release-8-4-1.html

[26] Apache HTTP server. [Online] http://httpd.apache.org/

[27] Apache Module *mod_proxy* [Online].
http://httpd.apache.org/docs/2.2/mod/mod_proxy.html

[28] Apache Module *mod_proxy_balancer* [Online].
http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html

[29] Ehcache [Online]. http://ehcache.org/

[30] Memcached [Online]. http://www.memcached.org/

[31] Cao, P. and Liu, C. *Maintaining strong cache consistency in the world wide web*. IEEE Transactions in Computing., 1998. 47(4): p. 445-457.

[32] Gwertzman, J. and Seltzer, M. *World-wide web cache consistency.* In *Proceedings of the 1996 USENIX Technical Conference, San Diego, CA.* January 1996.

[33] US Online Retail Forecast, 2010 To 2015 [Online].
http://www.forrester.com/rb/Research/us_online_retail_forecast%2C_2010_to_2015
/q/id/58596/t/2

[34] JGroups [Online]. http://www.jgroups.org/

[35] Maredia, R. *Automated application profiling and cache-aware load distribution in multi-tier architectures*. MSc Thesis. McGill University, Montreal, Quebec, Canada, 2011.