DESIGN AND DEVELOPMENT OF AN EFFICIENT PEER-TO-PEER WEB CACHE DISTRIBUTION SYSTEM

by

Garima Mahadik
B.C.A. (Honours), Devi Ahilya University, 2003

A PROJECT SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
Fall 2006

APPROVAL

Name: Garima Mahadik

Degree: Master of Science

Title of project: Design and Development of an Efficient Peer-to-Peer Web Cache Distribution System

Examining Committee: Dr. Janice Regan
Chair

Dr. Jiangchuan Liu, Senior Supervisor
Assistant Professor, Computing Science

Dr. Ramesh Krishnamurti, Supervisor
Professor, Computing Science

Dr. Jian Pei, SFU Examiner
Assistant Professor, Computing Science

Date Approved: November 30, 2006

Abstract

Traditional web caching systems based on client-server model suffer from various limitations like single point of failure, congestion, higher document retrieval time, limited cache space, etc. In this project, based on existing research on peer-to-peer communications, we develop an efficient peer-to-peer cache distribution system, in which all the peers share their respective web caches, joining the individual web caches to create a huge virtual cache space. We implement an efficient algorithm for managing and searching in the combined cache. We also develop a peer-to-peer communication and messaging protocol to enable interaction and communication among the peers. Finally we implement consistency control to prevent dispersion of old, unoriginal web objects in peers' caches. Our experimental results demonstrate that the prototyped system has a high performance.

To my family

Acknowledgments

First and foremost, I would like to thank my Senior Supervisor, Dr. Jiangchuan Liu for his invaluable guidance, support and readiness to help throughout my stay at the graduate school and also for his enthusiasm and encouragement while evaluating my work.

I would like to thank my Supervisor, Dr. Ramesh Krishnamurti, for providing insightful comments and helpful suggestions that helped me improve the quality of my work. I would also like to thank Dr. Jian Pei for agreeing to examine my project and devoting precious time to review my work.

A special acknowledgement goes to Michael Steger for his continuous support and understanding that helped me overcome the difficulties on the way. I also thank all my wonderful friends for their help and motivation.

Last but certainly not the least, I am very grateful to my parents and family for their colossal and unbroken support during all the education years.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Peer-to-peer networking has become a prominent topic of research recently. Caching web contents has the potential to benefit from peer-to-peer technology. Web caching is a widely-deployed technique to reduce the latency observed by web browsers as well as to reduce the load on web servers. In a network, data access is always an 'expensive' operation. By keeping frequently accessed data in the cache and not releasing it after the first use, we can avoid the cost and time required for the reacquisition and release of the data. This results in greatly improved web performance and better user experience, since we do not have to contact the server every time we need the data. With peer-to-peer networks, we can extend web caching to 'shared web caching' where the machines share their cache contents. Shared web caches are increasingly used in organizations (corporations, universities, government agencies, Internet Service Providers, etc.) where web clients are connected directly to the shared cache via a high-speed connection. Because many such organizations are connected to the Internet over low-speed, congested links, the average response time for the requested web objects can be substantially reduced if the objects are in the organization's shared cache. Moreover scalability is another benefit of using caching in peer-to-peer networks. Since cached data is accessed across multiple sessions and web applications, caching can become a big part of a scalable web application design.

1.1 Motivation

Many network communication protocols are based on the traditional client-server model. With a server listening for requests, clients connect to it and ask for services. However, this

model tends to intensify the burden on dedicated servers. This illustrates the importance of and need for another model: The *'peer-to-peer model of communication'*. Based on this model, data exchange relies on individual computers' computing power and storage capacity to distribute the load in a self-organizing manner. Each node functions both as a server and a client. Thus peer-to-peer computing is emerging as a distributed computing technology that enables direct resource sharing of both computing services and data files among a group of clients over the Internet.

With the expansion of the Internet's resources and users, the implementation of web cache servers has brought a great improvement in the retrieval time of web files. These cache servers store copies of web objects which may be accessed frequently by the client, thus reducing the need of retrieving a popular object from the original web servers. Internet proxy caching is a representative example of client-server based caching. Proxy caching is an effective solution to quickly access and reuse the cached data on the client side and to reduce Internet traffic to web servers. A group of networked clients connects to a proxy cache server. Given a web request from a user, the browser first checks if the file exists in its local cache. If so, the request will be served from that cache. Otherwise the request will be sent to the proxy cache. If the requested data object is not found in the proxy cache, the proxy server will immediately send the request to an upper level proxy cache, or to the web server without considering if it exists in other browsers caches. We believe that the proxy server needs to be restructured to exclude this default behaviour consideration. The possibility of a proxy cache miss which is a browser cache hit may have been considered low although no such study has been found in literature. It is desirable to understand the potential performance gain by sharing data among browsers. We have the following qualitative arguments for it. First, since browser caches are not shared among themselves, the size of the proxy cache is limited, which is a big drawback. Also if no data consistency is maintained between browser caches and the proxy cache, it is certainly possible that a data object is stored in one or more browser caches, but has been replaced. Finally, the number and types of web servers have increased and will continue to increase dramatically, providing services to a wider range of clients with time. Thus, the number of unique file objects cached in client browsers has increased and will continue to increase. It is impossible for proxy caches to cover all multi-requested file objects of different types.

Hence we see that web caching systems' design space is huge and building a good caching system involves several issues. So the challenges raised are: Is there an efficient way

to utilize the individual browsers' caches? Can we use the peer-to-peer paradigm in this context of web caching? How do we maintain the data consistency of the system in such a framework?

1.2 Contributions

This project work aims to answer all the questions raised previously. We study the problem of web caching in the Internet domain and use the peer-to-peer framework to achieve efficient and reliable content distribution. We exploit the advantages of peer-to-peer systems in conjunction with web caching to reduce network traffic and user latency. The contributions of this project work are as follows:

- We develop an efficient peer-to-peer framework which lets all the peers share their respective web caches, joining those individual web caches to create a large virtual cache space.
- We implement algorithms to manage and perform searches in the combined cache.
- We propose and develop a new peer-to-peer communication and messaging protocol to enable interaction and communication among the peers.
- We implement a data consistency mechanism to ensure that the cached files are up-to-date with the files on the origin server.
- We do a comprehensive performance evaluation of the system in a distributed environment. Based on real-world configurations, we implement the functionality for individual machines in the network to act as peers and communicate with one another. Thus, all the client machines cooperate in a peer-to-peer manner to provide the functionality of a web cache. Our performance results show that the peer-to-peer web cache content distribution system facilitates mutual sharing of web objects among end hosts with a high hit rate and low latency and is scalable for large networks.

1.3 Project Report Organization

The remainder of the project report is organized as follows.

- In Chapter 2 we provide background information for the rest of the report and present an overview of related work.
- Chapter 3 describes our system, the inherent algorithms used for managing and searching in this system and the development of a new peer-to-peer communication protocol for our application.
- In Chapter 4 the detailed system implementation of our work along with the design of the different modules is shown.
- The experimental results done in a distributed environment of several machines are shown in Chapter 5.
- Chapter 6 concludes the report and identifies some possible future development issues.

Chapter 2

Background and Related Works

This chapter provides background information for the remainder of the report. Section 2.1 reviews the main concepts of caching. Section 2.2 describes the related work done in common web caching protocols as well as their advantages and drawbacks. Section 2.3 gives a general overview of existing peer-to-peer systems in terms of their approach to caching and content distribution.

2.1 Concepts and Terminology

A **Cache** is a collection of duplicated data where the original data is stored somewhere else or computed earlier. The principle of caching is implemented in a wide range of applications within today's IT infrastructure. This reaches from small segments of high-speed memory within CPUs to solid-state cache memory in hard drives to web browsers and other internet applications. A simple definition of cache is: a temporary storage area where frequently accessed data can be stored for rapid access.

Caching works as follows: when the cache client (a CPU, web browser, operating system) wishes to access a data object, it first checks the cache. If a matching entry is found, it is retrieved from the cache and recorded as a cache hit. For example a web browser program might check its local cache on disk to see if it has a local copy of the contents of a web page URL before fetching it over HTTP from the web server.

The alternative situation, when the cache is consulted and found not to contain the desired object, is termed as a cache miss. The data is then fetched from the original source

and passed on to the requesting client, the result returned stored in the cache, ready for a future access.

However, the value of caching is greatly reduced if cached copies are not updated when the original server data changes. Cache consistency mechanisms ensure that cached copies of data are eventually updated to reflect changes to the original data. There are several cache consistency mechanisms currently in use on the Internet: time-to-live fields, client polling, and invalidation protocols [1]. Time-to-live fields are a prior estimate of an object's lifetime that are used to determine how long the cached data remains valid. Each object is assigned a time to live (TTL) and when the TTL elapses, the data is considered invalid; the next request for the object will be to its original source. TTLs are very simple to implement in HTTP using the optional "expires" header field specified by the HTTP protocol standard [2]. The challenge in supporting TTLs lies in selecting the appropriate timeout value. Frequently, the TTL is set to a relatively short interval with the result that data may be reloaded unnecessarily, but also that stale data is rarely returned. TTL fields are most useful for information with a known lifetime, such as online newspapers that change daily.

Client polling is a technique where clients periodically check back with the server to determine if cached objects are still valid. It is based on the assumption that young files are modified more frequently than old files and that the older a file is the less likely it is to be modified. Adopting these assumptions implies that clients need to poll less frequently for older objects.

Invalidation protocols are required when a stronger notion of consistency is desired; many distributed file systems rely on invalidation protocols to ensure that cached copies never become stale. Invalidation protocols depend on the server keeping track of cached data; each time an item changes the server notifies caches that their copies are no longer valid. One problem with invalidation protocols is that they are often expensive. Servers must keep track of where their objects are currently cached. Invalidation protocols must also deal with unavailable clients as a special case. If a machine with the cached data cannot be notified, the server must continue trying to reach it, since the cache will not know to invalidate the object unless it is notified by the server. Finally, invalidation protocols require modifications to the server while the other protocols can all be implemented at the level of a web-proxy.

After studying the different consistency methods, in order to ensure consistency in our peer-to-peer framework, we follow an approach similar to client polling which periodically checks the freshness of cached files. More details on the consistency mechanism are given in Chapter 4.

To measure the effectiveness of caching, different performance metrics exist. The two most important performance measures for caching are Hit Rate and Latency.

- **Hit rate:**
  The percentage of accesses that result in cache hits is known as the hit rate or hit ratio of the cache. Thus

$$\text{Hit rate} = \frac{\text{TotalHits}}{\text{TotalHits} + \text{TotalMisses}} \times 100$$

  HitRate is an important measure of the efficiency of a caching system.

- **Latency:**
  In general, latency is defined as the period of time that one component in a system is waiting for another component. Latency, therefore, is wasted time. For example, in accessing data on a hard disk, latency is defined as the time it takes to position the proper sector under the read/write head. In networking, latency means the amount of time it takes for a packet to travel from source to destination. In VoIP terminology, latency refers to a delay in packet delivery- VoIP latency is a service issue that is usually based on physical distance, hops, or voice to data conversion. In caching, latency is the time it takes for the requesting client machine to receive a document and all its associated files directly from the host specified in the URLs for those objects. Thus we see that latency is an important measure of the capacity of the system. A reduction in latency means faster document retrieval from the system.

Web Caching is the caching of web objects (such as HTML documents and related data) for later retrieval. A web cache stores copies of documents passing through it; subsequent requests may be satisfied from the cache if certain conditions are met. Web caching provides three significant advantages:

1. Improvement in access performance: Performance is improved because redundant data-transfers are avoided.

2. Reduction in latency: The access latency is reduced because the request is satisfied from the cache instead of the origin server.

3. Reduction in server load: The server has to handle fewer requests since the web caches satisfy as many requests as possible.

Web caches can be deployed in a variety of ways. User agent caches, such as those in web browsers, are private caches, operating on behalf of a single user. Intermediaries can also implement shared caches that serve more than one person. Web caching is mostly deployed using proxy servers, the points through which the clients of a large network access the internet. Hence since World Wide Web(WWW) caches are often implemented as proxies, web caching is often referred to as proxy caching. As shown in Figure 2.1, a proxy cache application sits between the web server and a client and forwards the request to the server on behalf of the client. It saves copies of these responses (also called representations) for itself. Then, if another request comes at a later time for the same URL, it can use its saved response instead of contacting the origin server again.

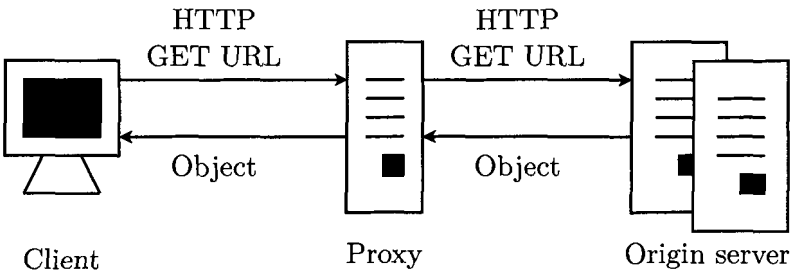

Figure 2.1: Web caching with a proxy server.

However the underlying model supporting the transmission of web objects is still very much based on the traditional client and server model. In this regard, existing web caching systems suffer from some drawbacks. In conventional web caching systems (based on a client-server model) the stand-alone proxy [2] sits between client and server. But a stand-alone proxy is a bottleneck; there is a limit to the number of clients that can use the same cache and thus the effectiveness of the cache is limited. Thus with the increase in demand, it becomes increasingly difficult for a single proxy to handle all the requests and

hence scalability becomes an issue. A caching proxy also represents a single point of failure. Finally such systems are bounded by the cache space size.

In order to serve an increasing number of clients, there have been several proposals for grouping together proxies [3] to achieve better performance but they still have overhead concerns [4]. In a client-server paradigm the problem of a single node (possibly server) failure remains and the proxy servers are always faced with an insufficient disk space. This is where the advantage of peer-to-peer networks is exploited.

A peer-to-peer computer network is a network that relies on the computing power and bandwidth of the participants in the network rather than concentrating it in a relatively low number of servers. This provides a mechanism for sharing and delivery of user-specified files among groups of people who are logged onto the peer-to-peer network. An important goal in peer-to-peer networks is that all clients pool their resources together. As a result, when more nodes arrive and demand on the system increases, the total capacity of the system also increases. This is in contrast to a traditional client-server model wherein adding more clients could mean slower data transfer for all users. Also the distributed nature of peer-to-peer networks increases robustness in case of failure. If a single peer node fails, it does not lead to a drastic reduction in system performance and the network can recover quickly. Thus, we see that applying web caching or content distribution via peer-to-peer network will allow machines to function as a distributed storage medium.

2.2 Web Caching Protocols

To implement shared web content distribution in a peer-to-peer framework, we decided to study a range of shared web cache protocols. We studied the Internet Cache Protocol [5], Hash Routing Schemes [6] and Cache Digests [7].

2.2.1 Internet Cache Protocol (ICP)

The Internet Cache Protocol (ICP) [8, 5] is a lightweight protocol used for communicating between web caches at the application layer. It runs on top of the User Datagram Protocol/ Internet Protocol (UDP/IP). An ICP message contains a 20 byte-header plus a variable-sized payload; the payload typically contains a URL.

ICP promotes sibling cache communication mechanisms, i.e. mechanisms to query nearby caches for a given document. It works as follows: Initially a web client sends an

HTTP request [2] to one of the shared caches. If the shared cache cannot satisfy the request, it simultaneously queries all the other sibling caches for the object. This querying is done via ICP messaging. If at least one of the siblings has the object, the shared cache requests the object(using HTTP [2]) from the first sibling cache that responds with an ICP hit message. Upon receiving the object, the shared cache stores a copy of the object and sends the object to the client. If none of the siblings has the object, the shared cache fetches the object from the originating server, caches a copy and delivers the object to the client. After studying ICP, we gather some of its advantages and disadvantages.

1. Advantages:

   (a) ICP eliminates the single point of failure problem. An important property of ICP is that there may be more than one copy of any given object in the sibling caches. This property creates a mirroring effect and popular objects are stored across all the sibling caches. So if one or more caches is unavailable, the clients will still be able to retrieve popular objects from the other caches in the network.

   (b) ICP serves to provide some form of load balancing. This is because an idle cache can turn around the replies faster than a busy one. Hence if a cache does not have the requested object, it can send a miss message and take on the next request.

2. Disadvantages:

   (a) Simultaneously querying all the sibling caches generates undesired additional traffic in the network.

   (b) Certain quantitative analyses [6] shows that ICP protocol generates additional network traffic (i.e. ICP messages) which must be processed by the caches. With ICP, a shared cache processes many more ICP messages than HTTP messages.

2.2.2 Hash Routing Schemes

Hash routing schemes are another approach to implementing shared web caches to reduce unnecessary traffic during searching among the peer caches. Several hash routing schemes [6, 9] have been proposed.

A simple hashing scheme works as follows: All the clients store a common hash function[1] which maps URLs to a hash space. The hash space is partitioned into sets, with each set in the partition associated with one of the sibling caches. When a client desires an object, it first hashes the object's URL and then requests the object from the appropriate sibling cache. If the cache contains the object, it is returned to the client. If the cache cannot satisfy the request, it retrieves the object from the originating server, keeps a copy in its local cache and forwards the object to the web client.

The hash routing schemes form the basis of Cache Array Routing Protocol (CARP) [9]. It is a distributed caching protocol based on the known membership list of proxies and a hash function for dividing the URL space among these proxies. The hash space is partitioned among all the sibling caches. It uses a hash function to decide which cache to forward the request to. When a request is to be sent out, the protocol takes the URL requested and feeds it through the hash function to determine which member should be the receptacle of that particular URL request. An HTTP client agent which uses CARP can allocate and intelligently route requests for the correct URLs to the specific member cache thus eliminating duplication of cache contents and improving global cache hit rates. We enumerate some advantages and disadvantages of the hash routing protocol.

1. Advantages:

   (a) With the hash routing scheme, the problem of wasting storage space is solved because at most one copy of an object resides in the collection of shared caches.

   (b) Hash routing schemes can minimize the load on network traffic because the requests are not flooded to all the sibling caches but only to the one that is responsible for this object. Also due to this, the targeted object can be located more easily.

2. Disadvantage:

   The hit probability drops dramatically and instantaneously if one of the caches fails. This happens because of the lack of more than one copy of every requested object in the system. Hence when a cache, responsible for a set of objects fails, the objects

[1] A hash function is a deterministic mapping which maps inputs to a hash space. This results in random output hash values scattered over the hash space. Any change in the input values creates possibly a large change in the output hash values obtained [6].

will not be found at other caches till they are locally cached by the requesting caches. Hence due to the limited number of object copies in the system, the system is more vulnerable to single node failure.

2.2.3 Cache Digests

The Cache Digest scheme [7] aims to solve the problems of latency and congestion associated with protocols like ICP.

Cache Digests, unlike most of the other inter-cache communication mechanisms, support peering between shared caches without a request-response exchange taking place. Instead of the exchange, a summary of the cache (called the Cache Digest) is fetched by others which peer with it. Using Cache Digests, it is possible to determine with a high degree of accuracy whether a given URL is cached at a particular server or not. To determine this, we feed the URL into a hash function which returns a list of bits to test against in the Cache Digest. In the implementation of Cache Digests, URLs are concatenated with their corresponding HTTP method numbers (by which the URLs are requested) and the results are used to create a public key using the MD5[2] hash function [10]. These keys are then looked up in the Cache Digest. Given a public key of a URL, if the corresponding object is available in the cache, all bits in the Cache Digest associated with each of the indices of the public key should be set. This protocol has its own benefits and drawbacks.

1. Advantages:

   (a) Cache Digests reduce the network traffic. For instance, if we have a set of loaded caches, the inter-cache communication can use significant amounts of bandwidth. Each request to one cache produces a series of requests to the neighbouring caches causing some server load as well. With Cache Digests, however, the load is reduced. The Cache Digest is generated periodically and the transfer of digests thus happens not too frequently. So the other shared caches do not need to query the summary of others every time a new request is generated.

   (b) The Cache Digest compresses a large amount of cache information into a small array of bits.

[2]MD5 (Message-Digest algorithm-5) is a widely used cryptographic hash function which processes a variable length message into a fixed length output hash-value of 128 bits.

2. Disadvantages:

   (a) The algorithm is inherently imprecise. The resulting digest may contain some false hits for URLs which the cache does not actually contain. This happens because the list of objects is only updated on a periodic basis. So the cache might potentially expire an object soon after downloading the summarized index.

   (b) It is not possible to delete a single public key from the digest without rebuilding it from scratch. Removal of expired cache objects is only possible when the whole digest is recalculated.

After studying the advantages and disadvantages of the prevalent protocols, the system architecture is built making use of some of the advantageous ideas in them.

2.3 Overview of peer-to-peer systems

This section explores several relevant issues in peer-to-peer systems [11, 12, 13, 14, 15, 16, 17, 18, 19] in the context of content distribution or caching.

As mentioned before, peer-to-peer systems are distributed systems with all the nodes equal in functionality. They possess some special characteristics:

- Peer-to-peer systems share resources by direct exchange instead of requiring a centralized server system. In peer-to-peer systems, some degree of centralization is permissible for some specific tasks such as bootstrapping but for a routine operation, server dependence is not allowed. Thus nodes aggregate their individual resources and satisfy user requests independently in the peer-to-peer system without any central server intervention.

- Peer-to-peer systems are robust to variable connectivity. They have an ad-hoc nature and peers join and leave the network at their own discretion without direct control of any entity. This ad-hoc nature promotes robustness to variable connectivity.

- They are cost-effective because they utilize existing resources and do not require additional infrastructure.

When it comes to the peer-to-peer architecture model, we should note that the peers form an overlay network[3] among each other. These overlay peer-to-peer systems can be structured or unstructured. The categorization is based on whether the overlay network is created in a non-deterministic or ad-hoc manner as nodes are added (unstructured), or whether the creation is based on specific rules (structured).

2.3.1 Unstructured networks

Unstructured peer-to-peer networks [11, 12, 13] are loosely coupled i.e. the placement of files is independent of the structure of the overlay network. In an unstructured network, contents need to be located. The searching mechanism ranges from brute force methods like flooding to slightly more sophisticated approaches like random walks. Some examples of unstructured networks include Napster [17], Gnutella [18], Kazaa [19] etc. For example Napster is a hybrid decentralized file-sharing application in the sense that peers join a network and perform search function by contacting a central server but the actual content distribution is done by the peers themselves.

2.3.2 Structured networks

Structured peer-to-peer networks [11, 12, 13] are tightly coupled i.e. the files (or pointers to them) are placed at precisely specified locations in the overlay network. These systems provide a mapping between contents (file-identifiers) and the locations (node addresses) by using hash-based routing like Distributed Hash Tables (DHT) [20, 21].

A hash table is a data structure that associates keys with values. Its primary operation is an efficient lookup: given a key, find the corresponding value. It transforms the key using a hash function into a hash value, a number that is used by the hash table to locate the desired value. Distributed Hash Tables are a class of decentralized distributed systems that partition the ownership of a set of keys among participating nodes and efficiently route messages to the unique owner of any given key. Some examples of structured DHT-based peer-to-peer architectures include CAN [14], Chord [15], Pastry [16], etc.

Table 2.1 presents a comparison of the important differences between the structured and unstructured networks.

[3] An overlay network is a logical network built on top of another network. A virtual link can be established between two peers in an overlay network without a direct physical link between them in the underlying physical network.

Table 2.1: A comparison of structured and unstructured peer-to-peer networks

Unstructured Networks	**Structured Networks**
They can tolerate to a high degree the transient nature of peers.	It is hard to maintain a proper overlay structure when nodes join and leave at a high rate.
They support flexible queries containing keywords and regular expressions.	They support only rigid query matching i.e. the query should contain the exact identifier of the file requested.
The searching process is expensive because the requested file can be anywhere in the system. However the distribution of contents across many nodes (for example servents in Gnutella network) makes it ideal for practical file-sharing applications.	They provide faster searching with less overhead by using DHTs, thus a good scalable solution for exact match queries.
There is a lack of guarantee of locating the searched file since the scope of search is restricted. In systems like Gnutella, the search is performed to a fixed number of hops (TTL).	These networks guarantee that if a file is present in the network and hashed to an identifier, it will be located.
Unstructured networks are useful when keyword searching is commonly used, there is a small network and a transient node population.	The structured approach finds use in highly scalable networks for efficient searching and retrieval.

There are some systems which can be termed as *'Loosely Structured Systems'*. Our peer-to-peer system falls under this category. The defining characteristic of a loosely structured system is that its nodes can predict which node is most likely to store a certain content. Thus they avoid blindly broadcasting request messages to all (or a random subset) of their neighbours. Also our network pools disk space in peer computers to create a collaborative virtual cache space. It is capable of handling peer join and peer leave operations very well. This is based on the assumption that every peer node knows at least one other peer node in the network. Hash-based query matching is used which ensures a faster search process in the peer-to-peer system. The system implemented can be used in highly scalable networks for efficient searching and retrieval.

Chapter 3

Peer-to-Peer Caching System, Operations and Peer-to-Peer Communication Protocol

This chapter gives the details of our peer-to-peer caching system. In Section 3.1 we define our problem objectives and later describe our peer-to-peer cache distribution system. Section 3.2 presents the algorithms used for handling key operations of our system like locating and retrieving the web contents. In section 3.3, we introduce a new peer-to-peer communication and messaging Protocol which enables interaction among the peers in the network.

3.1 Overview of the system

3.1.1 Objective

The primary design issues in peer-to-peer cache distribution are where to cache copies of objects (cache placement), how to keep the cached copies consistent (cache consistency) and how to redirect clients to the optimal cache (client redirection). Our system addresses these and other relevant issues by defining specific objectives. Our main objective is to reduce the shortcomings of a traditional client-server model for caching contents by using a peer-to-peer system to promote efficient caching of web contents thus leading to a higher hit rate and a lower latency in the network. Our implementation should provide an efficient way of utilizing the individual peers' caches for cache placement. It should provide a mechanism

for qualified client redirection. It should maintain consistent information in the respective caches and be scalable to a large number of nodes in the network while maintaining its consistent, high performance.

3.1.2 System description

As mentioned in the previous chapter, a peer-to-peer system differs from a client-server system. These differences lead to more challenges from an implementation perspective. First of all, the system should be self-governing and able to act independently. Peers cooperate to build a network without the intervention or provision of service from other systems. We have to make sure that every peer is aware of its neighbour peers and thus coordinates and interworks with other peers. Secondly in the peer-to-peer system, each node can act as a transient server and client. A requesting peer is a client and a chosen peer is the server. To take advantage of these inherent characteristics, we design our system by enabling every peer to collectively share its cache contents. Combining these individual peer caches, we create a huge virtual cache space which is at the peers' disposal.

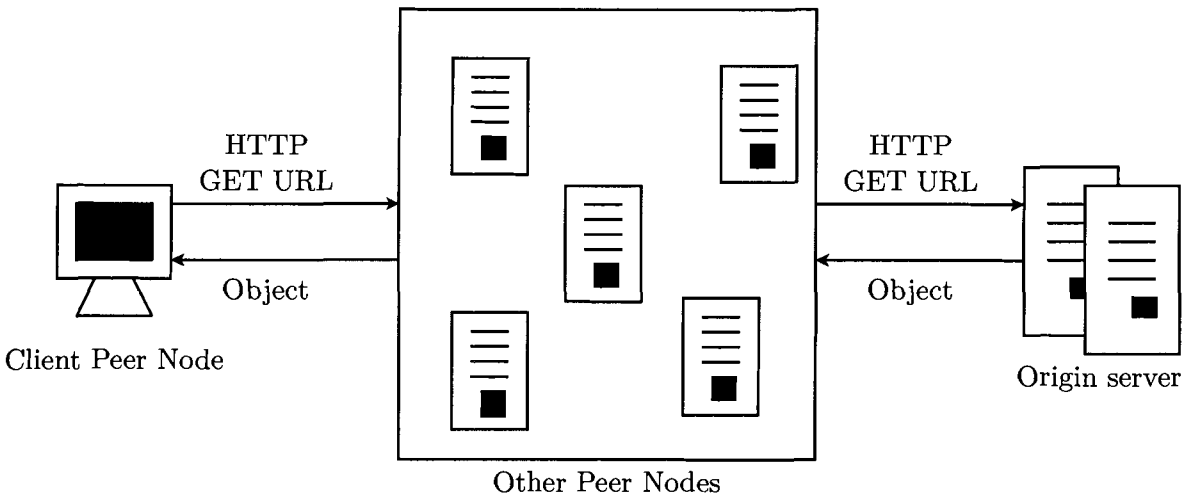

Figure 3.1: Web caching with a group of peer-to-peer nodes

This system design benefits all the peers. More requests are satisfied within the peer group since more web objects are cached nearby leading to less retrievals from the server.

After studying the benefits and drawbacks of the common protocols, we designed our system to make use of some of the advantageous ideas in them. We deploy an algorithm in which a hash-based scheme is used to compute the hash of the object URL. Also a key is assigned to each node in the shared cache. Using this hash output we determine which node to request by comparing the node key and the hash of the objects URL. Therefore every node in the network will be specialized to hold information on the objects with similar URL hashes. The hashing module which performs these two important hash operations in our system is shown in Appendix A. However unlike the hash-routing scheme mentioned above, we will allow multiple copies of objects in the shared caches. This approach disperses the caches and also reduces the risk of a single point of failure. The workload of retrieving popular objects will be dispersed among different nodes, improving the efficiency of our system. We assign the URL identifier(URLID) of objects to the live peer whose node identifier(NID) is the closest by comparing them. When a search within the peers is needed, the initiating node will first find a peer with a NID closest to the URLID of the requested object. A single search message is then sent to this peer with the belief that this peer has the greatest chance of caching the object or it may know which other node has cached the object. As only one search message is sent, the traffic load for each search is very low.

On the other hand, although only one peer is queried, we would not worry about the hit rate because now we make use of a prevalent protocol called the Probabilistic Search Protocol [22]. With this, apart from having a Local Directory (LD) or local cache, every peer has a Directory Cache (DC) that points to the presumed location of resources managed by other peers. Making use of the protocol in our system, each node maintains a directory cache containing the search history of the other nodes. Since a node when previously requested an object has a large chance of having cached it, a search request can be redirected to another peer with reference to this history. Thus with time, a node would have recorded a search history with many URLIDs close to its NID. and subsequent search requests will have a greater chance of hit. This approach involves fewer hops for search messages and results in a high hit rate. In our system, as we would like to further limit searching time, the number of redirections according to search history is limited to one. Thus this approach reduces wasteful traffic in our network as each time only a node with a NID closest to the hash of the object name is requested. More details will be presented in Section 3.2.2.

3.2 Key Operations

3.2.1 Initializing a peer-to-peer web caching system

The initialization phase consists of:

1. Configuring an initial neighbour list for every peer
2. Maintaining the neighbour peer table

As is implied by 1, one requirement of the peer-to-peer web caching system is that a new node entering the system should know at least one other peer node to create an initial neighbour set. So the system is given an initial list of cached addresses to start up. This input is manual at startup. For every node, we have a list of node addresses in the network and when a new node wishes to join the network, it will try to connect to the nodes specified in the list to join the peer network.

After the initial configuration is taken care of, the peer-to-peer network needs to expand and every node needs to increase its neighbour peer list. Thus step 2 above is performed using two basic approaches. In the first approach, to allow a peer to know more neighbours around it, it tries to query its neighbouring peers for more peer information. Since a peer-to-peer network is a dynamic network with the nodes joining and leaving the network occasionally- when a lot of neighbours leave the network, a node might be left with too few neighbours to allow for an efficient searching of web objects. In such a case, it enlists the help of its neighbouring peer nodes to again expand its neighbour peer table. Algorithm 3.1 conveys the approach for getting more peer information.

Algorithm 3.1 Peer Request Algorithm

1. For a node, set a lower limit for the number of peer neighbours n_p.
2. If number of neighbours $< n_p$ then query neighbouring peers for more information.
3. The queried peers return a list of nodes to the original node.

Another approach is to use advertisement which will be very similar to the 'HELLO' message to initiate a connection. This is used in the protocol implementation discussed later. A node can sometimes be the advertisement sender and sometimes the forwarder of the advertisement. To prevent an advertisement from bouncing in the network, some sort of

control on forwarding is needed for forwarding. For this a TTL[1] count is used. If the TTL of a message is still greater than one when a node receives it, it will forward it otherwise not. The steps taken for peer advertisement are shown in Algorithm 3.2.

Algorithm 3.2 Peer Advertisement Algorithm

1. Original node broadcasts to other online nodes within the TTL hop count.
2. Other online neighbouring nodes will forward this advertisement to their peers.
3. The nodes receiving this will reply to the original node.

These mechanisms allow a node to eventually know more nodes in the peer-to-peer network. Similar services are also provided through Peer Discovery Protocol [23, 24].

3.2.2 Searching in a peer-to-peer web caching system

To strike a balance between the different approaches looked at in Chapter 2, we decide to make use of a hash routing scheme in our algorithm. Hashing was chosen because the output of a hash function can be very different even if the inputs are similar. This disperses different files with similar names to different nodes in the network. However an essential difference between this approach and the other hash-routing schemes is that unlike the hash-based schemes [6, 9], this algorithm allows multiple copies of the objects in the shared caches. This is done when request is issued by a client to its shared cache node which might not be the targeted node containing the hash request. Hence when the shared cache node starts the search process and retrieves the object from the targeted node, it caches a copy locally to serve future requests for the same object. Thus, this mechanism prevents a single node failure from causing a drastic drop of hit rate. More details about this will be explained shortly.

Hashing is required at two points in our system: in generating the node identifier of a node and in manipulating searches of files. Both hash outputs are generated using the Secure Hash Algorithm (SHA) [25], a popular hashing technique. It is considered to be a successor to MD5 [10], an earlier widely used hash function. SHA hashing has been extensively employed in applications like copy prevention systems, digital signature standards and many

[1]TTL is short for Time-To-Live. It is an 8-bit field in the Internet Protocol(IP) header that specifies how many hops a packet can travel before being discarded/returned.

file-sharing applications [26, 16]. The SHA function takes as input a message and produces an output called message digest. This message digest value is unique to that specific message or document and is sometimes referred to as the *'fingerprint'* of that message or data. The code for hash.java module which hashes the node address and the file's URL is shown in Appendix A. This module takes the node address of a peer node and hashes it to produce a 16 byte node identifier. Similarly it takes the URL and generates a 16 byte URL identifier. Hence the search is directed to the node with the 'Node Identifier' closest to the requested URL's 'URL Identifier'.

The searching algorithm is shown in Algorithm 3.3. The best case of a search is that the client can immediately find a requested object in its local cache and returns it immediately to the browser; there is no need to search among its peers. The worst case of a search is that the client cannot find the requested object in its local cache and asks one of its peers for the object. That peer looks into its request search history and finds a matching node that had requested the same object in the past. It would then redirect the requesting client to ask that node. However if the requesting client does not get a response from that node also, then the client node needs to retrieve the web object from the original web server. In this case at most two peer caches are queried before the node needs to ask the original web server for the web object. Another important point is the rate of growth of request history at every single node. In our approach, we limit the redirection to a single peer node, thus ensuring less traffic in the network. However, the system may exhibit start-up latency as it would need to initially record more search history.

3.3 Peer-to-peer Communication Protocol

In this section, we present the peer-to-peer messaging protocol and explain all the protocol operation implementation details. The format of the peer-to-peer communication message in the protocol is as shown in Table 3.1.

Algorithm 3.3 Peer Searching Algorithm

1. Preconditions:
   - Assign a node identifier (NID) to every online node in the network using SHA-1 hash function [25].
   - Each node maintains a request search history of its neighbour nodes and updates it.
2. A peer cache node receives a search request from a client and hashes the URL to get the URL identifier (URLID).
3. It searches for the object (using its URLID) in its local cache.
4. `if` (*hit*)
   Go to step 10.
   `else`
   Go to the next step.
5. The requesting node then finds a neighbour node whose NID is closest to URLID and asks it for the object.
6. `if` (*neighbour node has the object*)
   Requesting node retrieves the object from neighbour, go to step 10.
   `else`
   Go to the next step.
7. The neighbour node looks into its directory cache to determine the request search history of its neighbour peer nodes.
8. `if` (*a matching node found*)
   Neighbour returns this node information to the requesting node, requesting node searches for the object at the matching node, go to the next step.
   `else`
   Requesting node retrieves the object from original server, go to step 10.
9. `if` (*requesting node finds the object at the matching node*)
   It retrieves the object from matching node.
   `else`
   It retrieves the object from the original server.
10. Requesting node sends the object to the client.
11. Exit.

Table 3.1: Message format for peer-to-peer communication system

Bit Position	Contents
0 - 3	Opcode
4 - 7	TTL
8 - 11	Length of Data
12 - 15	Length of URL
16 - (16 + length of data - 1)	Data
(16 + length of data) afterwards	16 byte hashed URL

Opcode specifies the type of the message. The various opcodes defined in the message are listed in Table 3.2. TTL is the time-to-live of the message. Also the message contains two types of data, the Data field and the URL field. The difference between the two is that the URL field is used to store the hash of an object's URL during searching or in file retrieval among peers whereas the Data field is used to store data required during advertisement or on other occasions apart from the URL. These other occasions will be explained later.

Table 3.2: Opcodes defined in the message

Number	Opcodes
0	HELLO
1	HELLO_RESPONSE
2	SEARCH_REQUEST
3	SEARCH_HIT
4	ASK_OTHER
5	SEARCH_MISS
6	GET_FILE
7	GET_OK
8	GET_FAILED
9	REQUEST_PEER
10	REQUEST_REPLY
11	ADVERTISEMENT

There are four kinds of messages included in the protocol.

1. *Connection*: Messages for initiating and maintaining the connection include:

   - HELLO - This request is sent by a node when it first connects to the other nodes. After a connection is established, this message serves as a heartbeat message to notify others that the node is still online.

- HELLO_RESPONSE - This is a response to the HELLO messages received from the other peers. Upon receiving a HELLO message, a node will update the timestamp of the node sending the HELLO message to maintain its online status.

2. *Searching*: Messages for searching include:

   - SEARCH_REQUEST - This message is sent when a node searches for an object. To identify the requested object, it stores the hash of the URL in the URL part of the message and sends it to its peers. The TTL of the message is set to 1.
   - SEARCH_HIT - This reply is sent to a node sending a SEARCH_REQUEST indicating that the replying node contains the object the requesting node wants.
   - ASK_OTHER - This reply is sent to a node sending a SEARCH_REQUEST indicating that the replying node does not contain the object, but in its search history it found a node that had requested the same object before. Information of that previous node is contained in the Data part of the message.
   - SEARCH_MISS - This reply is sent to a node sending a SEARCH_REQUEST indicating that the replying node does not contain the object that the requesting node wants, nor can it find in its request history any previous node asking for this object. The requesting node must retrieve the object from the original source after receiving this reply.

   For the above messages, the node initiating the search will wait for the replies for a period of time. If the time limit is exceeded, the requesting node will immediately retrieve the object from the original source since no reply might indicate congested traffic in the network or that its peer nodes are offline. In both cases, further searching may have lower chances of a search hit or may require more time to find another node for searching. This increase in delay is undesirable and hence we avoid it.

3. *File-Retrieval*: File retrieval messages are:

   - GET_FILE - This message is sent when a node wants to get a file directly from a peer. The hash of the file's URL is specified in the URL part of the message. After a node receives an ASK_OTHER reply, it looks at the return host name in the Data part of the reply and sends this GET_FILE message to get the object from that node.

- GET_OK - This reply is sent to a node sending a GET_FILE message indicating that the replying node contains the object that the requesting node requested. After sending this message, the file transfer will take place.
- GET_FAILED - This reply is sent to a node sending a GET_FILE message indicating that the replying node does not contain the object that the sending node requested.

Again the node initiating the file retrieval process will wait for a period of time before retrieving the object from the original server.

4. *Discovery*: Messages for discovery include:

   - ADVERTISEMENT - This message is sent by each node periodically to other online nodes for advertising purposes. The radius of broadcast will be indicated in the TTL field of the message.
   - REQUEST_PEER - This message is sent to other nodes to request their peer information. A node will send this message if its number of neighbours falls below a minimum threshold set for the number of neighbours.
   - REQUEST_REPLY - This reply is sent to the node sending the REQUEST_PEER message. Upon receiving this reply a node will check if it lists any nodes not in its peer list and will add those nodes accordingly.

The state diagram in Figure 3.2 illustrates the search process at a node and the various messages involved. It shows what received messages will change the states of the system and also lead to what messages are to be sent out. The initial state indicates a node listening for requests. Upon receiving an HTTP request, it searches for the web object locally. If the object is found, it sends an HTTP OK response to the web client and goes to the end state. Otherwise the node will send a SEARCH_REQUEST message to a neighbouring peer which is closest in its hash value to the web object (URL) hash value. There are three possible replies from the neighbouring peer. If the reply is a SEARCH_HIT message, the neighbouring peer will also send the web object to the requesting node immediately and thus that node will respond to the web client with an HTTP OK response. If it is a SEARCH_MISS message, the requesting node has to fetch the web object from the server. If it is an ASK_OTHER message, the search is redirected to another matching node from the

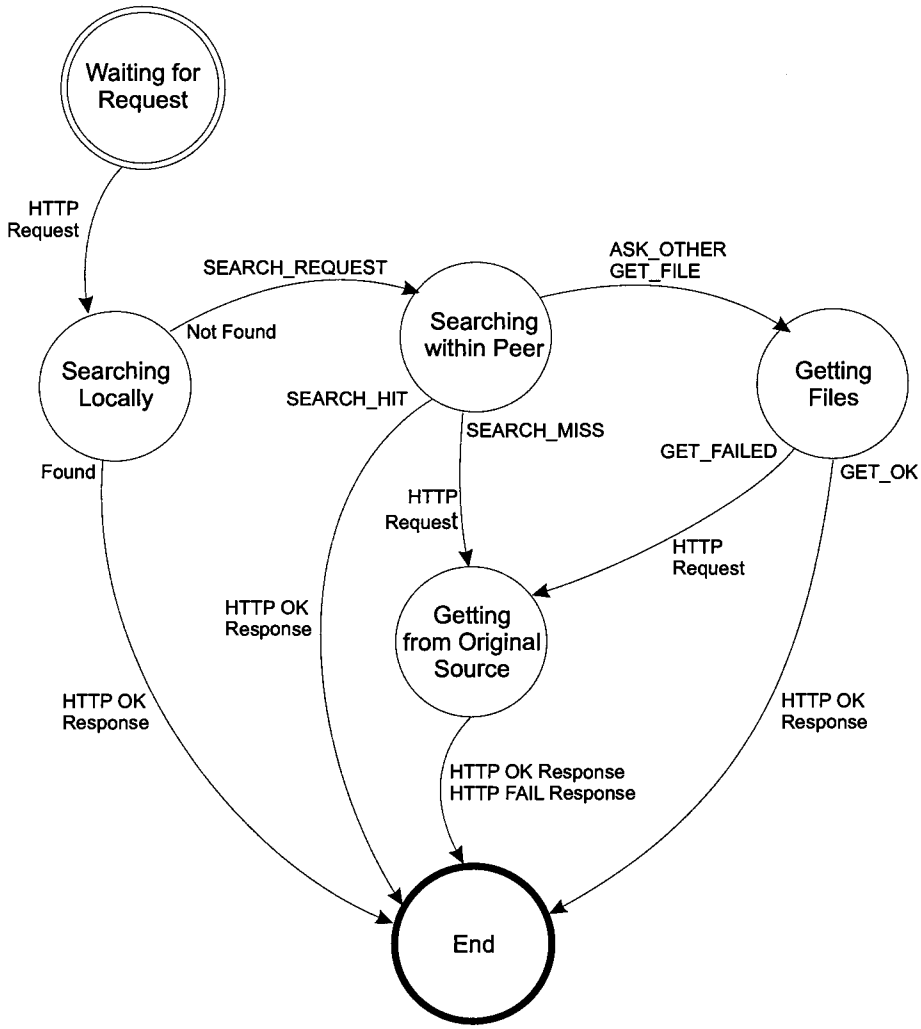

Figure 3.2: State diagram of a search operation performed at a node

neighbour's request history and the original requesting node will send a GET_FILE request to that matching node. At this point, there are two possible replies from that node. If the reply is GET_OK, the file is retrieved immediately from the matching node. If it is a GET_FAILED, the requesting node will retrieve the web object from the server.

3.4 Salient features of the system

Our system architecture and the peer-to-peer communication protocol have many advantageous features:

- Our search mechanism uses hashing to determine which node to ask. This reduces network traffic. Also the search time is reduced because only one node is queried for the search.
- We use a hash function that generates quite different outputs even though two input strings are very similar. This disperses different files from a single site to a different part of the network and prevents a single node failure causing a drastic drop in the hit rate of objects from a particular site.
- Our protocol also allows multiple copies among caches. This disperses content among caches and distributes the workload over all nodes.
- It eliminates the need to broadcast queries to all peers which is wasteful and time-consuming.

Chapter 4

System Modules

This chapter focuses on the practical implementation of the design outlines in the previous chapter. We discuss the different core modules and their respective functionalities in the peer-to-peer communication system. We give a basic structure of our implementation in Section 4.1 and elaborate on the different core modules in Sections 4.2, 4.3 and 4.4.

4.1 Basic structure

We have divided the system architecture into three main blocks: ***Program***, ***Physical File System*** and ***Main Memory***. The physical file system and the main memory support data storage for program operations. The data stored at each node includes configuration settings for program control, a list of addresses of cache peers, cached files and records of search history of peers. This data is essential for the program to run. We classify all the appropriate functions into one of the modules in these blocks for easier and clearer understanding of the whole architecture of this peer-to-peer system. We have implemented the system with a single program at each node.

Figure 4.1 shows the block diagram representation of the system implementation. We will elaborate this diagram throughout the chapter. We have chosen to use Java to implement our system and we have classified all the component classes into different modules. In subsequent sections, we describe the different components in the modules and their corresponding functions.

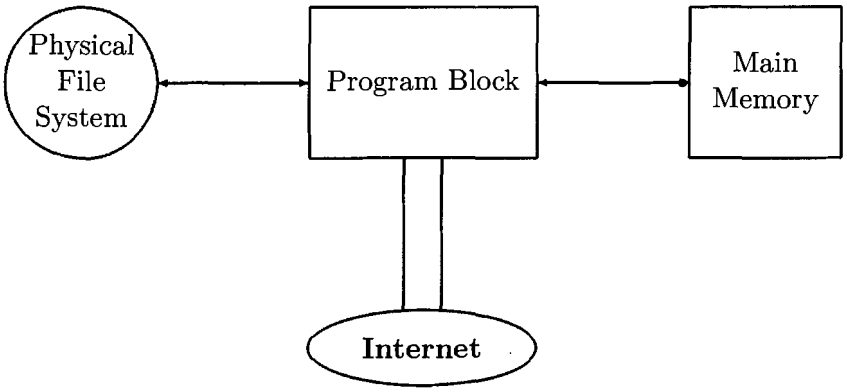

Figure 4.1: Initial System Architecture Block

4.2 Program Block

The ***Program*** block is the main system block entrusted with the peer-to-peer communication. Since every peer in the peer-to-peer network is a client and a server, each peer should have the ability of initiating connections as well as receiving connection from other peers. Each peer should be able to listen to HTTP GET requests from browsers, to search the cache and to carry out file exchanges among the peers. There are many more functions that a peer should be able to perform, including discovering new peers and advertising itself, maintaining communication with other peers, providing an option for users to change parameters dynamically so that the peers activity can fit different network environments, etc. To organize these functions, the program block is divided into six different modules:

1. Configuration Module
2. Listening Module
3. Search Module
4. File Transfer Module
5. Peer-to-Peer Connection Module
6. Consistency Module

Users interact with this Program block via a Graphical User Interface.

4.2.1 Configuration Module

The configuration module is responsible for reading and writing the system parameters from and to a configuration file. This module has a *Config* class which handles and records all the configuration parameters required by a peer. The parameters necessary for configuring the environment and for operation are listed in Table 4.1.

Table 4.1: Fields in the configuration file

Parameters	**Possible values**
Port	[1-65535]
HTTPPort	[1-65535]
CacheFilePath	[a valid path]
CacheAddressPath	[a valid path]
SearchTimeInterval	[positive integer]
HeartBeatInterval	[positive integer]
HeartBeatTimeOut	[positive integer] (default 1)
TimeToLive	[positive integer] (default 1)
MinNumOfNeighbours	[positive integer]
MaxNumOfNeighbours	[positive integer]
AdvertiseInterval	[positive integer]
InitialExpirationMinute	[positive integer]
InitialExpirationSecond	[positive integer]
MaximumExpirationMinute	[positive integer]
MaximumExpirationSecond	[positive integer]
MinimumExpirationMinute	[positive integer]
MinimumExpirationSecond	[positive integer]
ExpirationPeriodChangeRate	[positive integer]
FreshnessListCheckInterval	[positive integer]

The meaning of the different parameters is explained below:

1. *Port* - This stores the port number for exchanging peer messages at the node, i.e. the port for peer-to-peer communication.

2. *HTTPPort* - This stores the port number used for receiving HTTP requests from the web client.

3. *CacheFilePath* - This stores the path of a directory storing the cached files. It is a link to the physical file system.

4. *CacheAddressPath* - This stores the path of a text file containing the IP address and the port number of peer nodes. This address list is used for peer connection at the startup of the system.

5. *SearchTimeInterval* - This stores the time interval for each search operation. When this time limit is exceeded, the search is stopped and the node retrieves the web object from the originating server and immediately forwards it to the client.

6. *HeartBeatInterval* - This stores the time interval for sending the heartbeat to neighbour peer.

7. *HeartBeatTimeOut* - This stores the time interval for checking an inactive peer. In the system, each peer connection is associated with a timestamp. This timestamp for a peer is refreshed when a HELLO message is sent by this peer or a HELLO_RESPONSE message is received from this peer. If at the instance of checking, the timestamp is older than *HeartBeatTimeOut*, then that peer is considered to be inactive/offline.

8. *TimeToLive* - This stores the Time-To-Live of an ADVERTISEMENT message and hence it is a restriction on the radius of advertising.

9. *MinNumOfNeighbours* - This stores the number of minimum neighbours that should be connected to a node.

10. *MaxNumOfNeighbours* - This stores the number of maximum neighbours that can be connected to a node.

11. *AdvertiseInterval* - This stores the time interval for sending ADVERTISEMENT messages to the neighbours.

12. *InitialExpirationMinute* - This stores the minutes part of initial expiration time for a cached object.

13. *InitialExpirationSecond* - This stores the seconds part of initial expiration time for a cached object.

14. *MaximumExpirationMinute* - This stores the minutes part of maximum expiration time for a cached object.

15. *MaximumExpirationSecond* - This stores the seconds part of maximum expiration time for a cached object.

16. *MinimumExpirationMinute* - This stores the minutes part of minimum expiration time for a cached object.

17. *MinimumExpirationSecond* - This stores the seconds part of minimum expiration time for a cached object.

18. *ExpirationPeriodChangeRate* - The value of this parameter determines the factor by which the expiration period should be increased.

19. *FreshnessListCheckInterval* - This stores the interval between checks of the freshness of a file from the freshness list.

The Config component of the peer will be responsible for reading the configuration parameters from the configuration file to control the behaviour of the peer.

4.2.2 Listening Module

The listening module is responsible for listening to HTTP requests from web clients and passing the required data (the URL and information on the requesting web client) to the search module. The listening module is associated with an object of the class *RequestListener*. The *RequestListener* operates as an independent thread. It has a listening server socket waiting for TCP connections from the web client at a particular port. To allow interaction between the web browser on the client's machine with our peer-to-peer web cache system, the browser should configure its proxy server IP address and port to be the node's IP address and HTTP listen port. Once the listener receives an HTTP request for a certain URL object from the client, it will initiate a search to a peer, passing the URL of the object to the search module. To determine the URL of the HTTP request, the *RequestListener* extracts it from the request line of the HTTP request message. It passes this resource to the search module through an object of class *Peer* and this will initiate a search in the peer network.

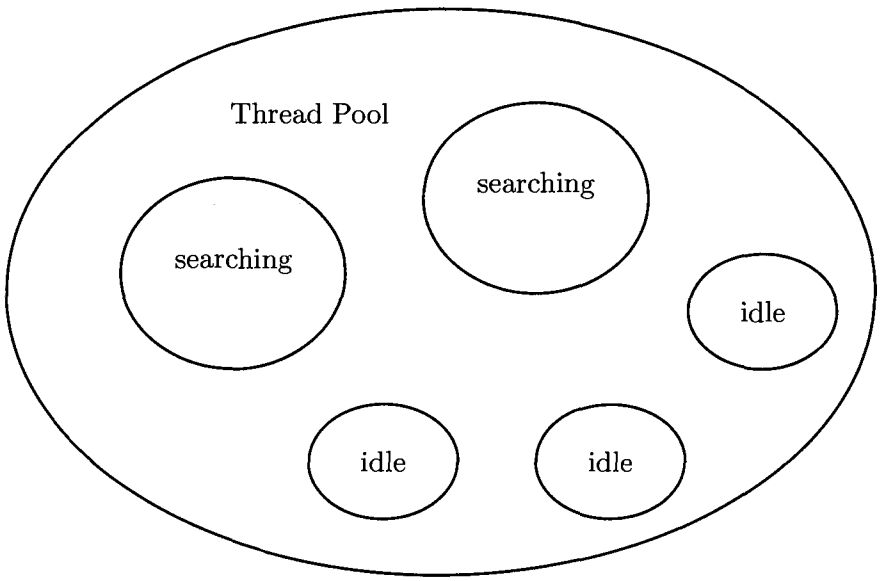

Figure 4.2: Illustration of encapsulating *SearchHandler* by a *SearchHandlerPool*

4.2.3 Search Module

The search module is responsible for searching a web object requested at its node. It is associated with an object of class *SearchHandlerPool* and this object associates with a vector of instances of *SearchHandler*. As the name suggests, *SearchHandlerPool* maintains a thread pool of Search Handlers (see illustration in Figure 4.2). It keeps track of the state of all threads in the pool. Whenever a new search is to be initiated, it retrieves an available thread to handle the search. *RequestListener* from the listening module then passes the information to the handler.

The *SearchHandler* is an independent thread. When activated and given the information for a particular search, it will maintain the state of the search according to Algorithm 3.3. Thus the *SearchHandler* will first try to find the targeted object in the nodes local cache folder. If no copy of this object is found, the node has to start a search between its peers. It will then ask a peer node whose NID is closest to the hash of the objects URL, and send the search request to this node. If the first search attempt is a hit, the *SearchHandler* will stop waiting and will get the file from the peer immediately through TCP file transfer. If the peer replies with a different node that possibly holds the object,

the *SearchHandler* will wait for a response from the peer whose IP address and port were provided in the reply message. Now the *SearchHandler* will wait for the reply from the redirected peer in a timed period. If no reply arrives during the specified wait period, or if it replies with a GET_FAILED message, the *SearchHandler* stops the search. In this case the file must be retrieved from the original web server, which is the job of the *HTTPHandler* in the file transfer module.

An important observation is that the search module is the module of the ***Program*** block which interacts with the other two blocks, ***Main Memory*** block to access the request history and ***Physical File System*** block to access the file cache. These two blocks will be discussed in sections 4.3 and 4.4.

4.2.4 File Transfer Module

The file transfer module is responsible for file transfers between nodes and the server, peers and web clients. Thus this module handles two kinds of file transfers present in the system: file transfer among the peers, which is a simple TCP transfer, and the retrieval of files from the original source, which is an HTTP GET request of the web object via the HTTP protocol.

- The file transfer between peering nodes involves two sides. One is a transient server and the other one is the requesting client. To handle the TCP file transfer between the peers in the implementation, the task of the server is done by a *FileTransferHandler* in a *FileTransferPool* (implemented in the same way as the *SearchHandlerPool*). It connects to a *FileReceiver* at the client side. The *FileTransferHandler* requires the IP address and port of the destination host and also the name of the file to be sent. Since the *FileReceiver* on the other side needs to know the name of the file for verification, the first line of the input stream of the file transfer socket will contain the hash of the file's URL. The file transfer will take place if a peer replies to the other peer with a SEARCH_HIT or GET_OK message. Once the connection has been established, the handler will send the file to the receiver and its job is finished. The *FileReceiver* is just a thread waiting for a request. After receiving the file, the *FileReceiver* will check to see if there is a *SearchHandler* waiting for this file. If the corresponding *SearchHandler* is found, it will inform the *SearchHandler* that the file has arrived and its search will be stopped. If no *SearchHandler* is waiting for this file, the file is simply discarded.

- If the web object is to be retrieved from the originating server, it must then be sent to the web client afterwards. This process involves HTTP request and response messages only, but nothing in the peer network communication. Therefore, the whole job is given to a *HTTPHandler*. The *HTTPHandler* is again an independent thread and encapsulated by an *HTTPHandlerPool*. If HTTP file transfer is needed, an available handler must first be retrieved by the associated pool. The *HTTPHandler* retrieves the file from the source and sends it to the client.

4.2.5 Peer-to-Peer Connection Module

This module is responsible for handling all the communication among peers. It is associated with a large list of objects of different classes. Figure 4.3 shows how these objects interact with each other. The various components of the peer-to-peer connection module are:

1. *Advertiser* - This object is used by a node to periodically advertise its existence by sending ADVERTISEMENT messages to its neighbours. The neighbours will then forward the advertisement to their neighbours. The range of forwarding is given by *TimeToLive* in the configuration file.

2. *HeartBeatHandler* - This handler is used by a node to send heartbeat (HELLO message) to the neighbouring peers to maintain its 'online' status in the network. Every heartbeat is valid only within a certain time interval, given by the parameter *HeartBeatInterval*. So before the heartbeat expires, the next heartbeat should be sent. The handler is also responsible for checking the status of other nodes by determining if their last sent heartbeat is still valid or not. If the heartbeat of a node expires, that node is regarded as 'offline'.

3. *Node* - Each *Node* object represents a peer node in the network, keeping all its information including NodeIdentifier, IP address and online status.

4. *NetworkInfo* - All the *Node* objects are encapsulated by the *NetworkInfo* object. Thus it gathers all the peer information in the network.

5. *PacketReceiver* - This object is listening to and receiving all the communication messages from the neighbour peers. After receiving a message packet, the receiver will call an available *PacketHandler* to handle the message.

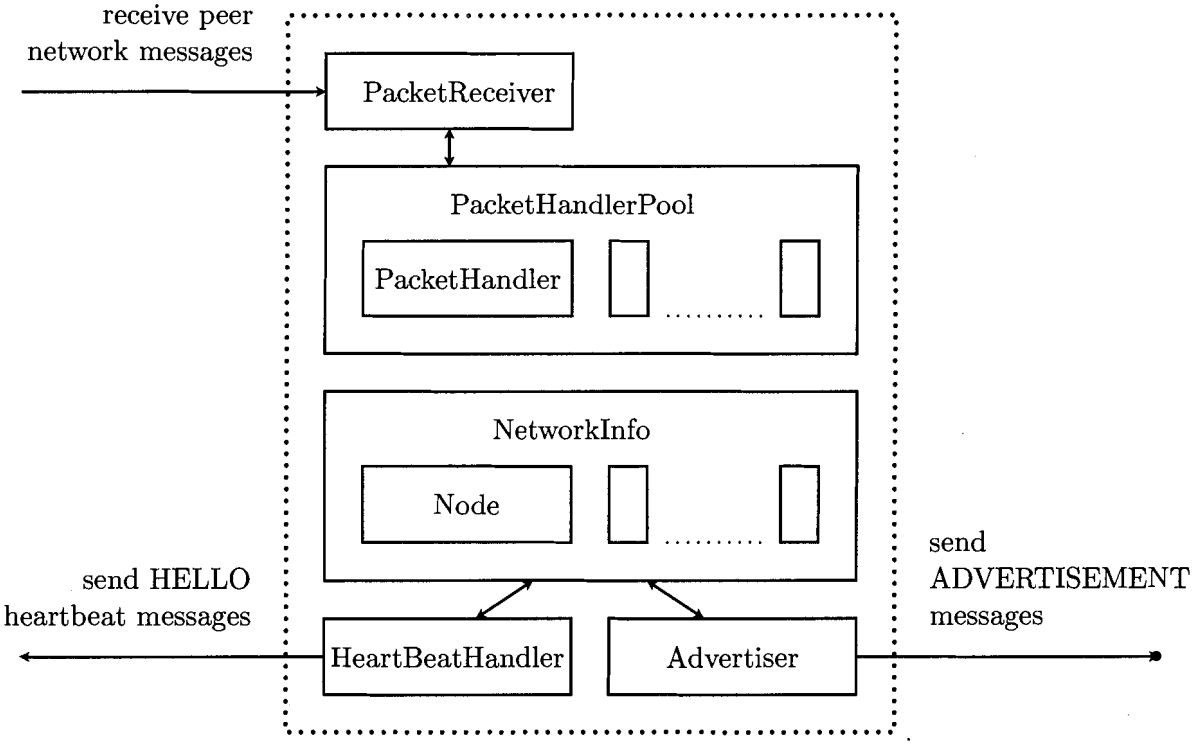

Figure 4.3: Overview of Peer-to-peer Connection Module

6. *PacketHandler* - This handler specializes in handling all the messaging among the peers according to the opcode specified in the message. The handler will look into all the fields in the message and take appropriate action according to our peer-to-peer protocol described in Chapter 3.

7. *PacketHandlerPool* - The *PacketHandler* objects are independent threads, so this pool acts as a thread pool of all the *PacketHandler* threads. Once the *PacketReceiver* receives a message, it asks this pool to return an available handler for this message.

4.2.6 Consistency Module

We also implement consistency in the system to ensure the retrieved files are up-to-date with the files on the original server. In Section 2.1, we briefly took a look at several techniques proposed for web cache consistency including TTL fields, client polling, invalidation protocols [1], etc. Here we adopt an approach similar to client polling; we assign a minimum expiration time and a maximum expiration time for a file.

The consistency module is responsible for checking the freshness of each file in the cache folder. It is implemented as class *Consistency*. In objects of class *Consistency*, instances of class *FileInfoList* and *HTTPUpdatingPool* are maintained.

The instance of the class *FileInfoList* is a file list in which each file is represented by an instance of *FileInfo* associated with an expiration timestamp. This list is sorted in increasing order of expiration timestamps and is likewise maintained. *Consistency* operates as a thread. Periodically (with the period specified by *FreshnessListCheckInterval* in the configuration module) the thread will extract the files whose timestamps have expired. It will then find an available *HTTPUpdateHandler* in the handler pool to check the freshness of the associated file. Again each *HTTPUpdateHandler* is an independent thread. Upon activation and having received files to check the freshness of, it will issue an HTTP request with the "if-modified-since" header field to the original server. If the server responds with status code 200 (OK), the file is updated. The expiration period is divided by the factor given by the *ExpirationPeriodChangeRate* value in the configuration module and the old file is replaced by the updated file from the server. However, if the server's response is status code 304 (NOT MODIFIED), the file is not updated and its expiration period is increased.

There is a lower and upper limit to the expiration period set by the *MaximumExpiration* and *MinimumExpiration* values in the configuration file. This ensures that there

is no wastage of resources if a file that is updated very frequently is still kept in the cache as this kind of object quickly goes stale. Once a file's expiration period is lower than *MinimumExpiration* value, it is removed from the cache. The upper limit on the expiration period guarantees that a file would not remain unchecked for infinitely long time.

4.2.7 Graphical User Interface

This module handles the display of information to users. There is a main frame drawn by the object of the class *GUI*. This allows users to connect and disconnect from the network. Moreover, it organizes lists of different network information. Users can monitor the traffic in form of tables. The traffic information is divided into different types: received and sent messages, online and offline connections, search messages, search history, freshness information of cached files, etc.

4.3 Physical File System

The physical file system provides storage for files. It includes:

1. File Cache: It is a directory in the physical file system. After each file retrieval, either from the original server or other peers, the file is cached. Here the 16 byte hash of the corresponding URL is used as the filename.

2. Address Cache: Each node holds a list of nodes in the network for connection. We call this node list the "address cache" of the node. It is a text file in the physical file system. Each line represents a node in the peer network. Each line is of the form "⟨IP address⟩:⟨port⟩".

3. Configuration settings: It is a text file in the physical file system. Each line represents a parameter and its value, and is in the form: "⟨parameter name⟩ = ⟨parameter value⟩".

4.4 Main Memory

Main memory has a faster access time but it is a volatile storage medium. It is used to store the search request history of all the nodes. Also we note that time is critical when

redirecting a search to another node, and thus the request history may not reflect an up-to-date relation between the peers and web objects after the system is restarted. Hence the search history is discarded every time after a system halt occurs.

The implementation of a request history at each node is an important part in handling searching. This request history stores all the requests that a node receives with the hash of the targeted objects' URL and the timestamps of the requests being recorded. In our system the request history is represented by a HashMap object with the hashed URL being a key and the key is mapped to a vector of strings containing the IP address and the port of the node issuing the request. The time stamp of the request is concatenated to the end of the nodes address, delimited by a ':' character. Once the node cannot find a file in its local cache storage, it will look into the request history, using the hash of the file's URL as a key to see if there is any host that had requested for the same object before. Recording the time stamp of the request is important because we use the policy that the node chosen to be asked for the second request should be the latest node that had asked for the object. The time stamp of a request will tell us how long ago a node has requested for the object. A new entry will be inserted into the request history every time the node receives a request, so the request history will keep on growing along the online period of the node.

The list of file information used in consistency checking is also stored in main memory. We first use a *FileInfo* object to store the information of all the files in the cache which have been directly downloaded from the original server or retrieved from peers. The information stored includes the filename, the original URL of the file, the period until this file expires and an expiration timestamp. A file is considered to be fresh before this expiration timestamp. We use a *FileInfoList* object to store a vector list containing all these *FileInfo* objects, with all the objects being sorted in ascending order according to their expiration timestamps.

In Figure 4.4 we present the **System Architecture Block** as updated with the design presented in this chapter.

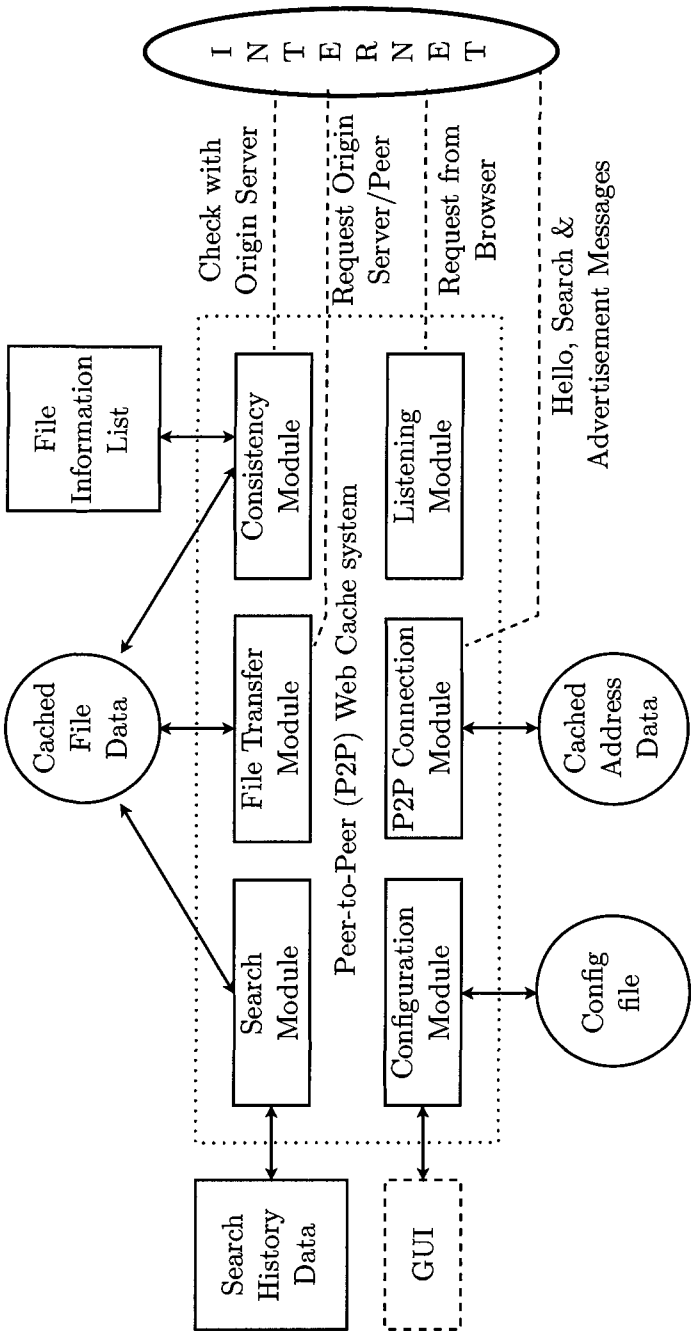

Figure 4.4: Final System Architecture Block

Chapter 5

Experiments and Results

This chapter describes the experiments undertaken to evaluate the performance of our peer-to-peer caching scheme. We compared and evaluated the design choices in terms of two performance metrics for caching: HitRate and Latency. We show how our system successfully achieves high hit rates with low latencies under different sets of conditions and different input parameters.

- **HitRate**: HitRate is the percentage of all accesses that are satisfied by the data in the peer web cache. It is also known as "hit ratio".
- **Latency**: Latency is the time taken between issuing an HTTP request by a client and receiving the requested file by the client. Unless otherwise mentioned, our calculated latency is for all the requests in one 'TimeSlot'. A single TimeSlot corresponds to a fixed number of requests. We try to diminish the effect of such factors as internet connection, document size, etc. by calculating this aggregated Latency so that the randomness caused by these factors can be mitigated. In our methodology, we do not constrain the latency to specific time units because the latency, in general, is dependent on many factors like the internet connection, the placement of the original server, etc which are not related to the algorithm and our implementation. We understand that the document retrieval times will be affected by these previous parameters and hence latency values may vary from one test environment to another. Our main purpose is to show the development of latency with our implementation, irrespective of the test environment constraints. Hence we scale the latency to 1 for every individual experiment. By scaling the latency, we maintain comparability between the different

experiments. We can compare individual points in different graphs with the same parameter values and see the variation in latency.

All the experiments were carried out with the nodes distributed in a network. The individual machines had 1600 MHz Pentium CPUs or equivalent and a range of 512 megabytes to 1 gigabyte of RAM and ran Microsoft Windows XP. Also, for each test, the neighbour list of every peer node was initialized to 2. After every test run, the peer caches were cleared for the next run. All the programs were written in Java.

5.1 Performance evaluation method

To evaluate the performance of our system, we design a method for issuing requests and starting the search for the requested files in our peer-to-peer web cache system. It calculates the performance data and stores them for future analysis and observation. Algorithm 5.1 shows this methodology.

Algorithm 5.1 Performance evaluation method

1. Assignment:
   - Construct Nodes.
   - Construct URLs.
2. Request: loop = 1 to Total_Number_Of_Requests
   - Pick a node randomly from the list of created Nodes.
   - Pick a URL randomly from the file of URLs.
   - Start the request.
3. Searching:

   `if` (*cached*)
   TotalHits++
   `else`
   TotalMisses++
4. HitRate = (TotalHits/(TotalHits + TotalMisses)) × 100
5. Latency = End Time - Start Time

To implement this method, we write an application which we call a 'Browsing Request Generator'. This essentially mimics multiple browsers by issuing requests to all the machines in the peer-to-peer system. The main application class 'Generator' makes use of two other helper classes: a 'Logger' class to log all the user data and performance metrics obtained and a 'Requester' class to request data from peer nodes. A useful enhancement we did was to make the 'Requester' class multithreaded. This is possible because the nodes themselves are multithreaded as well and thus capable of handling multiple incoming requests at the same time. This reduces the inherent idle time during the data transmission and reflects a real world behaviour where multiple clients can issue requests to a peer node at the same time. The 'Generator' class reads the node file and the URL file and initializes an instance of 'Logger' class with this information. It then creates an array of 'Requester' objects and starts a corresponding thread for every 'Requester' object. Each 'Requester' object in turn starts the 'Logger' instance to obtain a URL and a node to request the URL. Also since the peers are accessed as proxy servers, it sets the HTTP stack to a specific proxy host and port. For the test runs the peers are configured to transmit a flag, indicating whether the request is satisfied from within the peer-to-peer web cache system or retrieved from the original source. 'Requester' then sets the corresponding value in the 'Logger' class from where they are obtained later for future analysis. Figure 5.1 gives a depiction of our method.

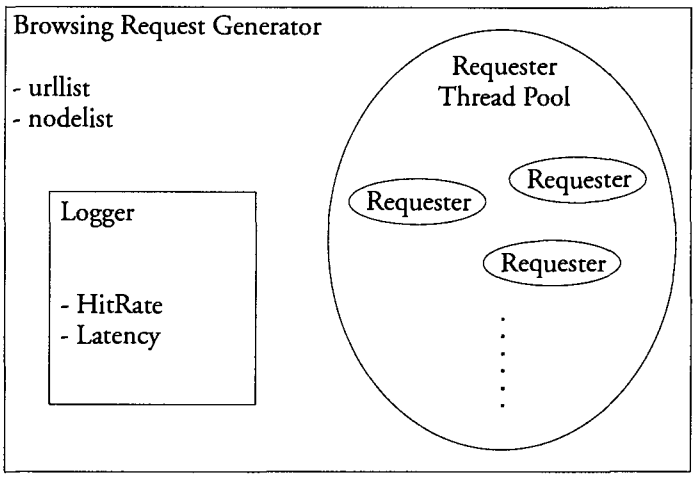

Figure 5.1: Browsing Request Generator

5.2 Experiments

5.2.1 Experimental Setup I

In our first experimental setup, the number of peer machines is varied from 2 to 8 while the number of URLs and the length of a TimeSlot is kept constant. As mentioned before, every single TimeSlot corresponds to a fixed number of requests. Table 5.1 shows the parameter set for this setup.

Table 5.1: Parameters used in setup I

Number of peer machines	Number of URLs	TimeSlot length
2	120	20 requests
4	120	20 requests
8	120	20 requests

Figures 5.2 and 5.3 show the variation of HitRate and Latency respectively for every TimeSlot. In all cases, the HitRate increases and eventually levels off at a very high value. The Latency, in turn, decreases with increasing TimeSlots. For simplicity and clarity in the graphs we stop showing the HitRate and Latency values after they reach steady values.

Observations:

We achieve persistent hit rates in the range of 90% to 100% with the system. Latency shows a decreasing trend, signalling a faster access after the initial build up phase of the cache. When we perform single trial runs, we observe noticeable variations in the performance. We attribute this to the randomness with which the URLs and the peer nodes are picked. At the points where dips are observed, the generator happens to select a different set of URLs from the URL file list which have not been precached. Hence we see a sudden decrease in HitRate pertaining to those TimeSlots. To eliminate this noise behaviour from our results, we conducted several single runs for every test case and calculated the average HitRate and average Latency for each TimeSlot. This is shown in Figures 5.4 and 5.5 together with standard deviations. We conducted 10 trial runs for every case and the figures show the final average obtained after the 10 runs. We see that the standard deviation decreases slightly with time with the hit rate for every single run approaching a high steady value. For all the subsequent experiments, we did 10 runs per test case and then showed the average.

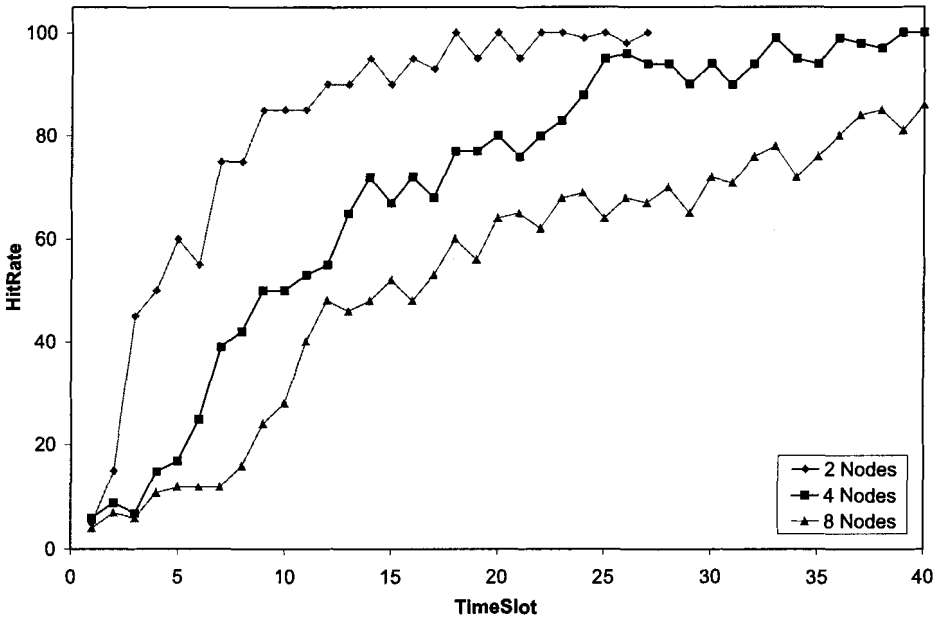

Figure 5.2: Setup I, HitRate with respect to TimeSlot

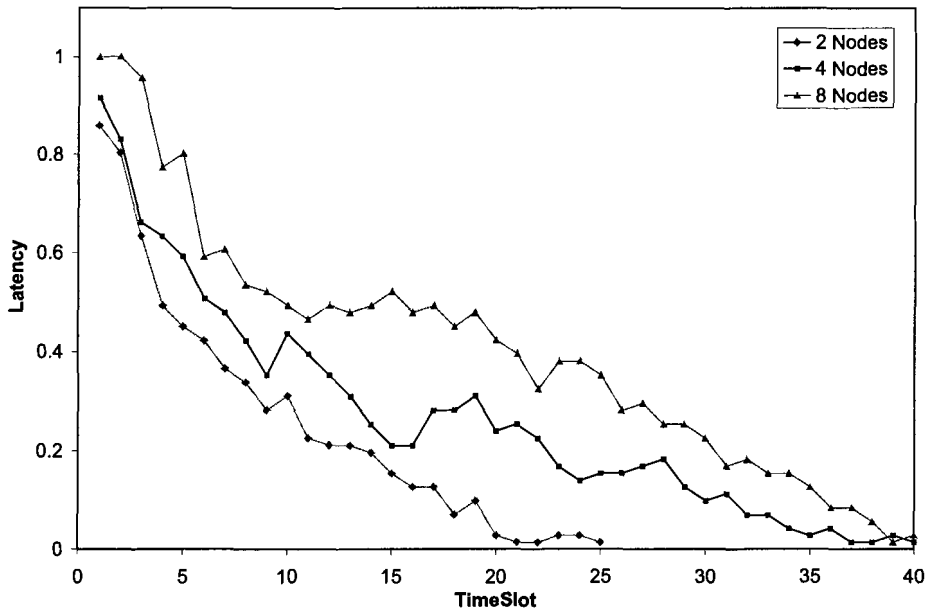

Figure 5.3: Setup I, Latency with respect to TimeSlot

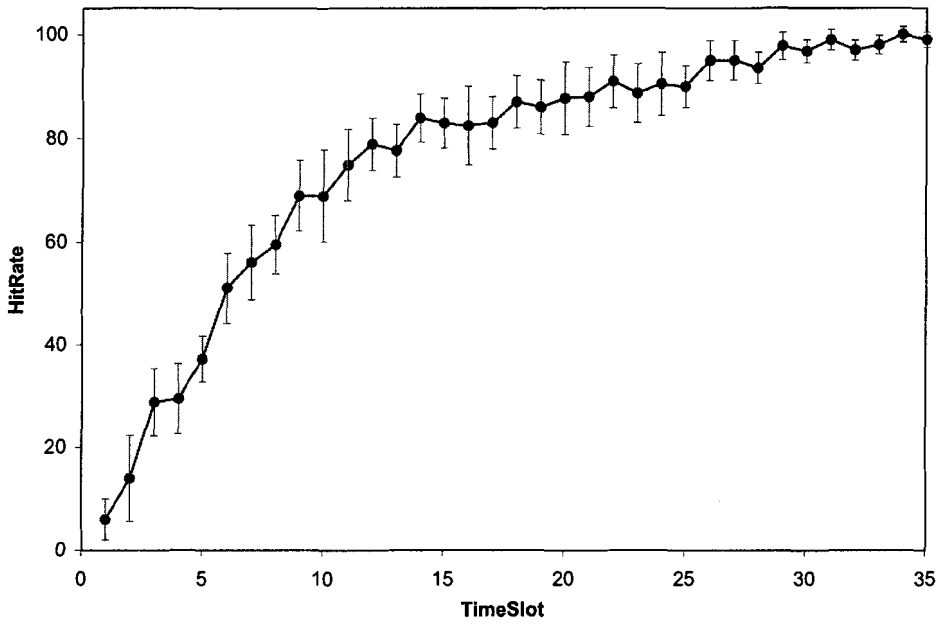

Figure 5.4: Average HitRate over multiple runs with respect to TimeSlot

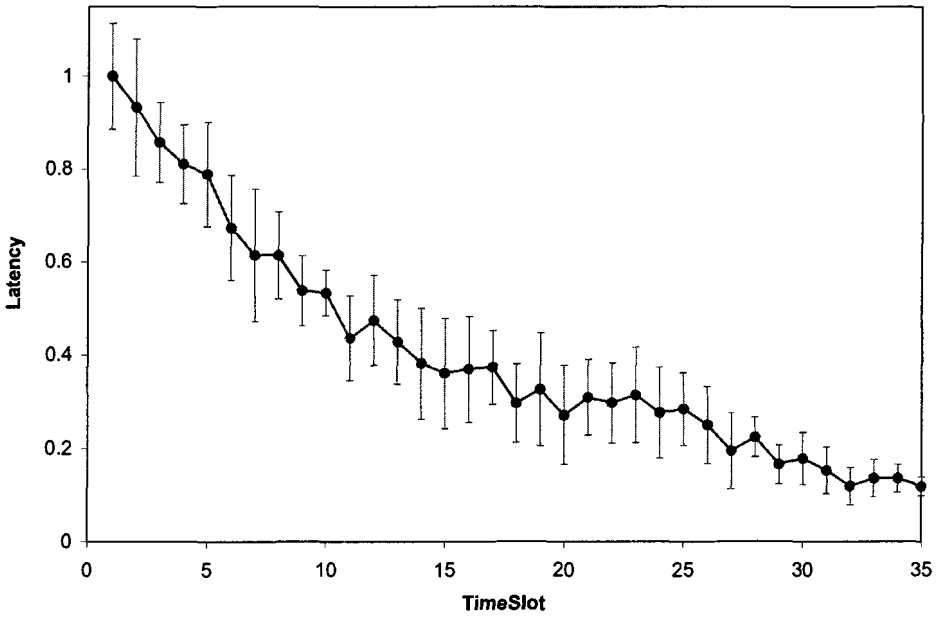

Figure 5.5: Average Latency over multiple runs with respect to TimeSlot

Another important observation is that although we achieve a high HitRate, the time taken to achieve this goes on increasing with the increase in the number of nodes. One reason for this is the setup where we issue a fixed number of requests (20 in this case) for every TimeSlot. As an example, we compare a single TimeSlot in the cases when we have two nodes to the case when we have 10 nodes. In the first case, each node may receive 10 requests and hence caching would be faster at their respective cache. However in the second case, each node may receive 2 requests on average and as a result, less caching occurs at those nodes. Another reason is that in a larger network, not every node knows every other node and thus the effective cache per node is smaller than in the case of fewer nodes.

This is the setup commonly investigated when measuring cache performance. However in the peer-to-peer network, every user gets equal preferences and equal access to resources i.e. every user is capable of handling the same number of requests and generating corresponding responses. To balance this situation, we developed a more practical second experimental setup in the following.

5.2.2 Experimental Setup II

To improve upon the shortcomings of the first setup, we modify our 'Browsing Request Generator' application to generate a variable number of requests corresponding to the network size to maintain a consistent load on every node. For the experiment, we effectively modify the length of the TimeSlot to be the previous length times the number of nodes.

With this modification, we have a randomly distributed consistent request load per node. We perform the experiment with the same parameters as experimental setup I along with this modification (shown in Table 5.2). Figures 5.6 and 5.7 illustrate the quality of our results.

Table 5.2: Parameters used in setup II

Number of peer machines	Number of URLs	TimeSlot length
2	120	20 requests
4	120	40 requests
8	120	80 requests

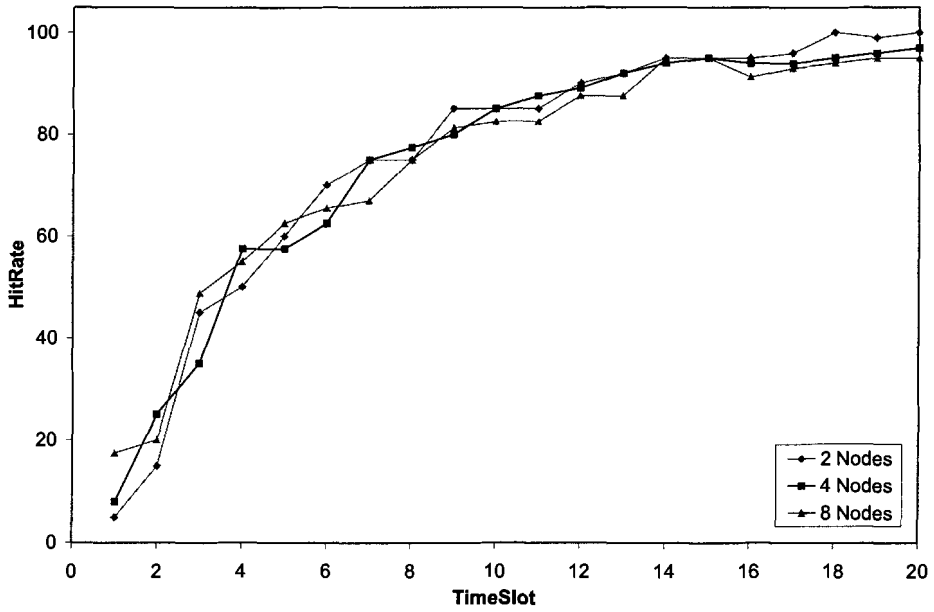

Figure 5.6: Setup II, HitRate with respect to TimeSlot

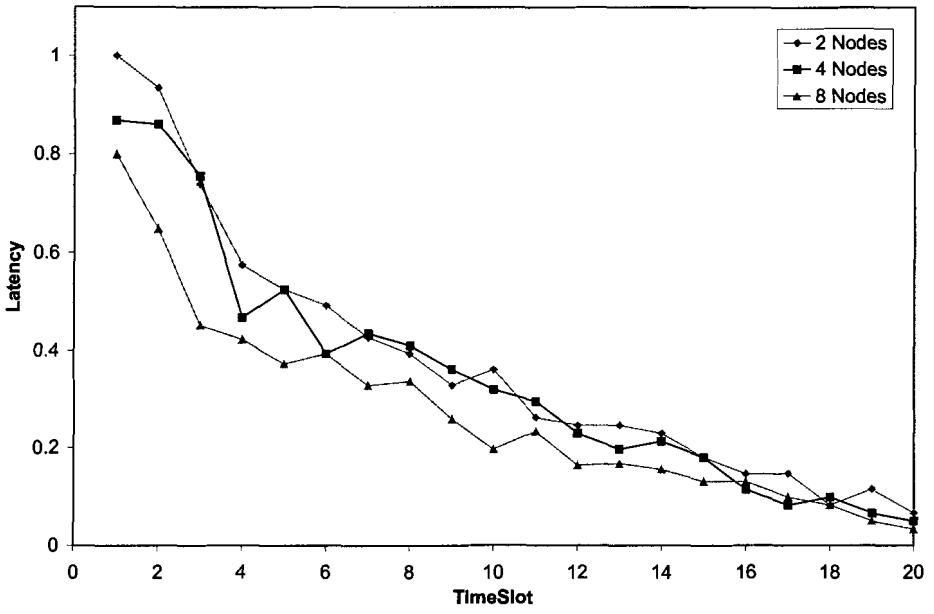

Figure 5.7: Setup II, Latency with respect to TimeSlot

Observations:

The results prove that our modifications are both useful and practical. We immediately see that even though the number of nodes is increased, the performance for both HitRate and Latency is not affected as in experimental setup I. This is because now we maintain the same number of requests per node, irrespective of the number of nodes in the system. Thus our system is capable of maintaining a high system performance with the increase in the network size and hence is scalable.

5.2.3 Experimental Setup III

In this setup, we do another experiment to see how our peer-to-peer caching network performs with an increase in the number of URLs. This is to test the behaviour of the system with increased randomness in user requests. The parameters used are summarized in Table 5.3. Figure 5.8 shows the variation of HitRate with the increase in the number of URLs and Figure 5.9 shows the corresponding variation in Latency.

Table 5.3: Parameters used in setup III

Number of URLs	Number of Nodes	TimeSlot length
40	4	20 requests
80	4	20 requests
120	4	20 requests
160	4	20 requests
200	4	20 requests

Observations:

Our results show that we succeed in achieving close to 100% HitRate. However, the time taken to achieve it goes on increasing with the increase in the URL list. We attribute this behaviour to the fact that since the number of nodes is fixed, we get more and more new URL requests (which have not been cached before) as the URL list goes on increasing and hence it takes more time to achieve the desired hit rate. Latency is variable in the beginning however it decreases and steadies out considerably towards the end.

In this context, we would like to briefly mention an important concept called the Zipf Law [30] which governs the user behaviour on the internet. According to this, a majority of users tend to visit a small number of popular sites or webpages. Numerous studies [31]

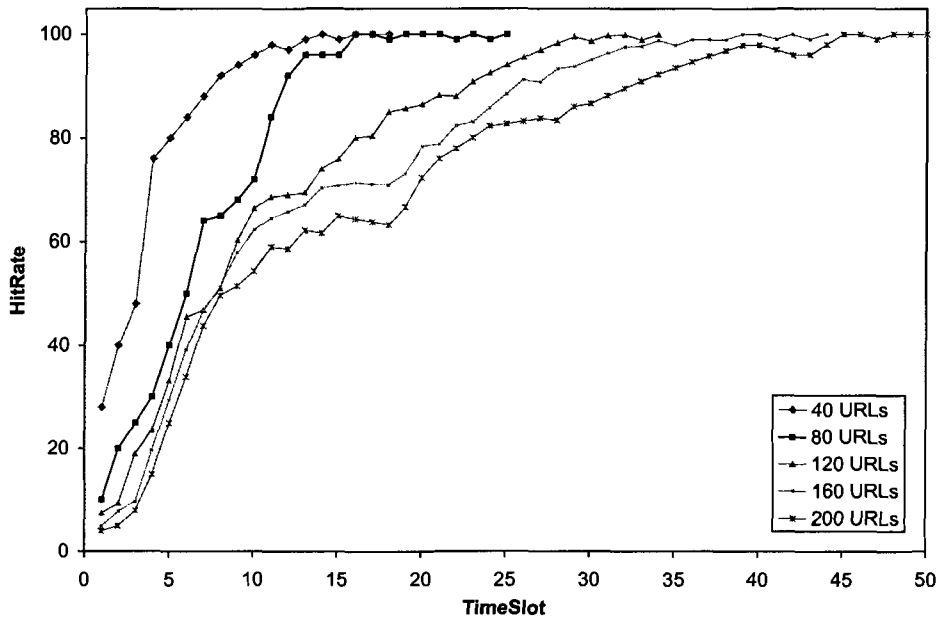

Figure 5.8: Setup III, HitRate vs. TimeSlot with varying number of URLs

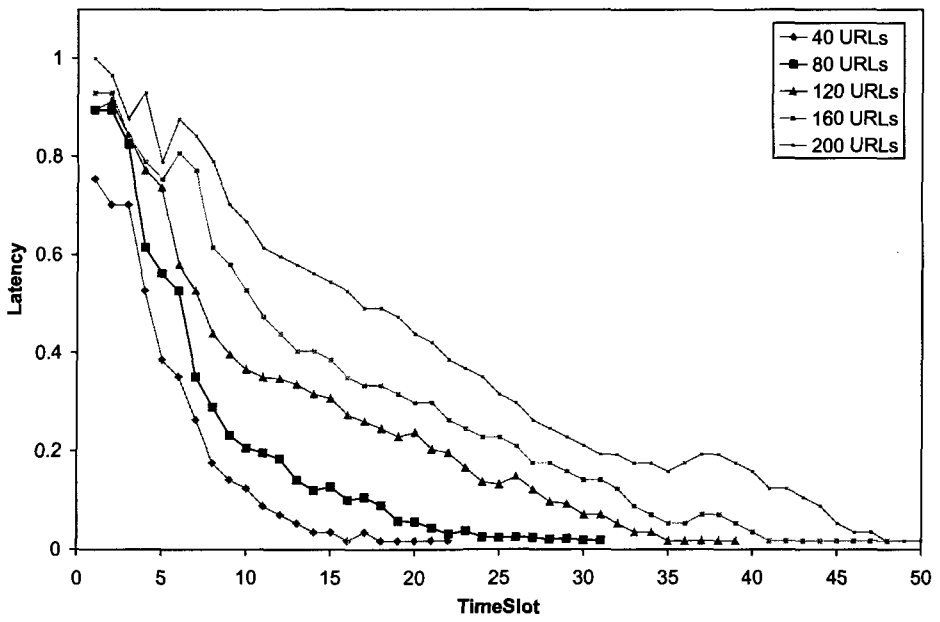

Figure 5.9: Setup III, Latency vs. TimeSlot with varying number of URLs

have found that the request distribution (the relative frequency with which web pages are requested) follows Zipf's Law. So we do not need to worry about the increasing number of URLs since we know that the number of 'different' user requests will typically be limited.

5.2.4 Experimental Setup IV

To examine how well our peer-to-peer web caching system achieves better performance as compared to a client-server model without cache collaboration, we calculated the HitRate and Latency in our test framework by using our model as well as by using a traditional client-server model supporting only local caching. Table 5.4 lists the parameter values used for this test environment. Figures 5.10 and 5.11 prove our results.

Table 5.4: Parameters used in setup IV

Parameter	**Value**
Number Of Machines	4
Number Of URLs	40
1 TimeSlot	15 requests

Observations:

We see that our method succeeds in achieving a higher HitRate as compared to the client-server model in comparable TimeSlots. As for Latency, we observe that there are some points where latency of our model is higher than client-server latency. This happens when a peer node has to query its neighbour peer node and wait for the search response which might cause a delay. However the advantage of our model is that the Latency is consistently decreasing and becoming steady as opposed to the other model where latency is widely fluctuating. Thus over a longer time period, our model performs better Thus promising good user satisfaction. The reason for the better performance of our model as compared to the other one can also be attributed to the fact that our load gets consistently distributed rather than all the requests going to one server which might get overloaded. Also another big advantage results from the addition of more nodes. With our model, more nodes joining means access to a bigger cache among all the peer nodes and little effect on HitRate with a single node failure. However with the other model, more clients requesting documents from the original server might lead to overloading.

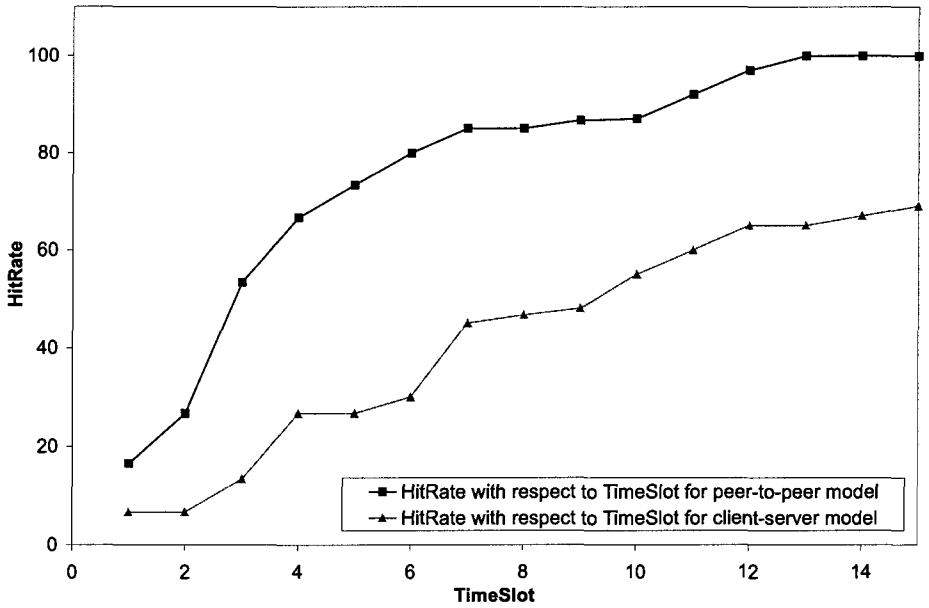

Figure 5.10: Setup IV, Comparison of models: HitRate with respect to TimeSlot

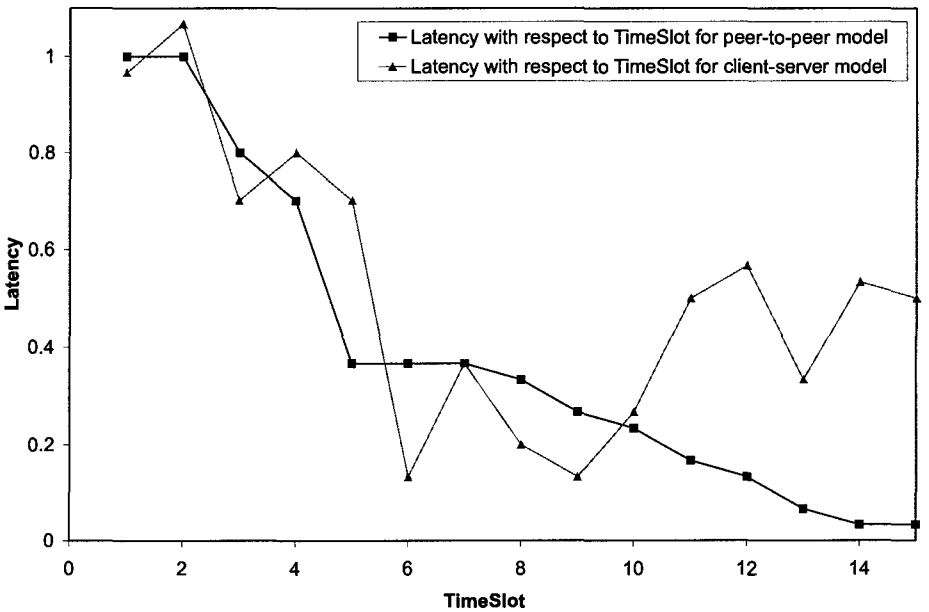

Figure 5.11: Setup IV, Comparison of models: Latency with respect to TimeSlot

5.2.5 Experimental Setup V

An important feature of a peer-to-peer network is the transient nature of peers. As discussed earlier, peer-to-peer networks allow joining and leaving of nodes without leading to network failures. Any peer can leave or join the network at any time. We also need to verify that the system is able to recover from these events and is capable of returning to and maintaining its high performance. We undertake experiments to perform node join and node leave operations.

In the case of a node leave event, we have initially four machines in the peer-to-peer network and a list of 120 URLs. After a certain runtime two peer nodes leave the network. Figure 5.12 shows how our system does a graceful and quick recovery from this performance degradation.

Likewise, we perform experiments to see how new nodes joining the network affect the performance metrics. For that case, we have an initial peer-to-peer network with two nodes. The nodes joining have existing nodes in their neighbour lists. Figures 5.12 and 5.13 show our results.

Observations:

Figure 5.12 shows how a node leave operation affects the performance of the system and the system recovers and returns back to normal. After the eighth TimeSlot, two nodes leave the peer-to-peer caching network. Hence at the ninth TimeSlot, we observe a sharp decline in HitRate. This is because the cache of the nodes leaving is no longer accessible. Hence requests which could be fulfilled by these nodes' caches will be misses. The system then starts to recover and achieves high hit rate as observed before. Likewise, for the Latency, there is an increase between eighth and ninth TimeSlot when the nodes leave the network. This happens because now more requests have to be satisfied from the original server. Hence the system can handle node leave events and come back to the previously observed behaviour for hit rate and latency.

Figure 5.13 shows how nodes joining the network affect the performance. Initially there are two nodes in the network and after the sixteenth TimeSlot, two more nodes join the network. We observe that the HitRate decreases and Latency increases after the new nodes join the network. This happens because the new nodes which join have initially an empty cache and the possible requests directed to them may be misses and may lead to

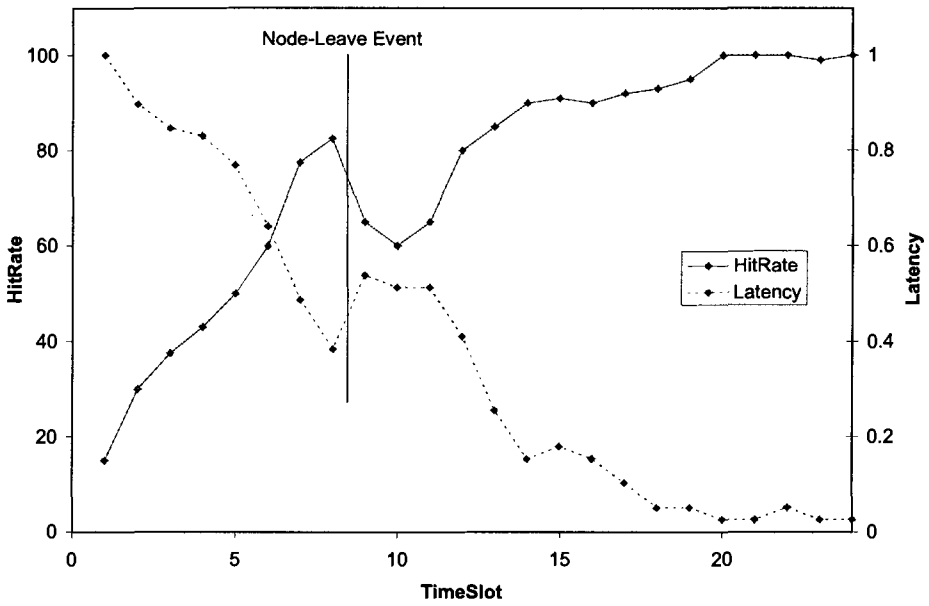

Figure 5.12: Setup V, Effect on HitRate and Latency with Node Leave Event

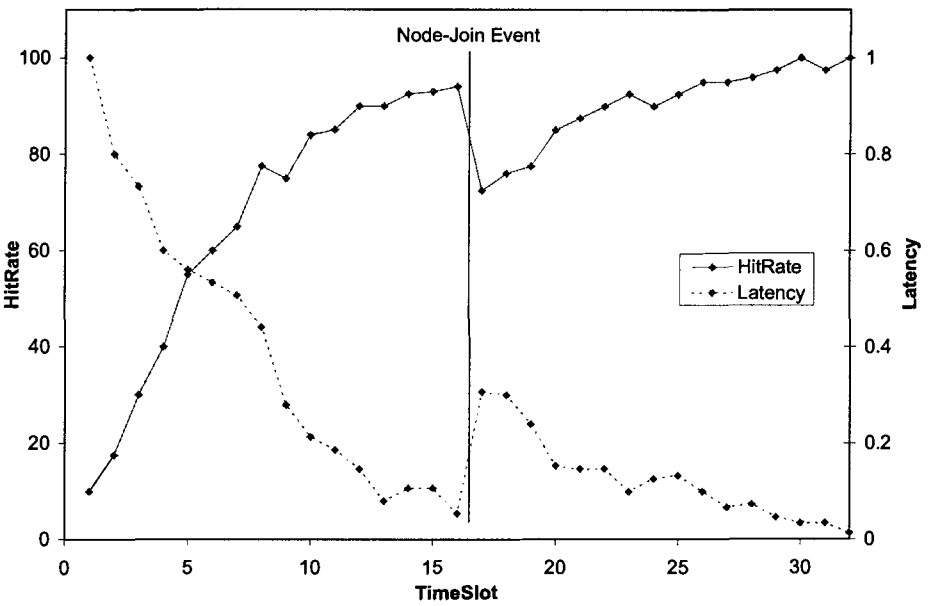

Figure 5.13: Setup V, Effect on HitRate and Latency with Node Join Event

object retrieval from the server. So the HitRate and Latency suffer initially. But after their cache is built up, we observe that in both cases the HitRate and Latency start following their normal trend. Thus, the system is capable to handle, recover and stabilize after nodes joining the network.

By analysing node join and node leave events in this experimental setup, we show that our peer-to-peer web cache system supports the peer-to-peer characteristics and does not suffer from a single point of failure.

Chapter 6

Conclusion and Future Work

The increasing popularity of the World Wide Web (WWW) presents many challenges such as increased network bandwidth usage and excessive document retrieval latency. In this project, based on existing research, we implemented a novel peer-to-peer content distribution system intended to reduce network latency and load. In this chapter, we summarize the project and discuss some future directions.

6.1 Summary of the Project

The World Wide Web (WWW) is a document distribution system based on the client-server model. Server systems tend to become slow when overloaded. Congestion can also occur at network exchange points. A common, yet expensive way to alleviate these problems is to upgrade the overloaded resource: a faster server, more network infrastructure etc. But this approach is not economically feasible for every application. Caching emerges as a good solution to reduce the latency experienced by end-user on the web. In this project, we applied the framework of peer-to-peer networks to increase and maintain a high system performance while providing high end-user satisfaction with increased hit rates and reduced latency.

We developed an efficient peer-to-peer framework which lets all peers share their respective web caches to create a large virtual cache space. Using the feature for content distribution of peer-to-peer networks, distributed storage capacity is utilized by our system.

We proposed and developed a new peer-to-peer communication and messaging protocol to enable interaction and communication among peers. We designed a protocol for

our system in which dispersion of web object copies are allowed between the participating caches. Another important feature that we designed is to keep a history of request on every node, storing all the information about previous search requests received by the node.

To evaluate the performance of our system, we conducted an experimental study by using machines in a peer-to-peer network. We observed and analysed the effects of different parameters like the network size and URL input size to determine the performance of the system using the metrics hit rate and latency. Our implementation produces results with high performance values. This implementation is an improvement because it not only satisfies an increasing number of clients in a peer-to-peer setting, it achieves this by resolving the problems of single node failure, overburdening of the server leading to degradation in user experience, long document retrieval times, etc.

6.2 Future Directions

There are several possible future enhancements to this project work.

- Development of a privacy framework: An interesting and important idea which comes to us is that of encryption of cached files for high user content safety. Currently if any node in the peer-to-peer network retrieves a file (either from server or other neighbour peers), the file is stored on its machine and the node can look at it or retrieve it later. But we wish to allow a framework so that these files are not accessible without the implementation software and so all the peers get access to the files only when they have the software installed. This prevents an outside user from looking at what requests were issued and satisfied inside the peer-to-peer network. This would in turn guarantee increased privacy to the users in the network.

- Development of a security framework: With the increased advantages of an open cache framework shared by all the nodes in a dynamic peer-to-peer system come the challenges of maintaining the reliability of such a system. In a peer-to-peer system, users rely on each others' contents 'believing' that the files they are getting from one another are of credence. However that might not be the case if we have some compromised nodes in the network storing fraudulent copies of web objects. To ensure credibility of cached data, a security framework needs to be developed. Work [32] has been done for maintaining trust and security in such a network. The

trust model developed is based on Subjective Logic [33][1]. Subjective logic operates on opinions which are the probability measures of belief, disbelief and uncertainty. The trustworthiness of a node in the network is determined by the other nodes' opinions (belief, disbelief, uncertainty) of this node. Thus the task of applying subjective logic to develop a trust implementation in our system should be addressed to check for and ensure authentic peer nodes and authentic cache information.

- Replacement policy: Considering the web caching scenario, a replacement policy is also a significant issue but it has not been dealt with in this work. If the cache has limited storage, it may have to delete an entry to make room for a new one. The heuristic used to select the entry to delete is known as a replacement policy. However in recent years, with the relatively small size of web objects and decreasing cost of disk space and memory, web caches are able to store most cacheable content and rarely need to remove contents. The consistency policy implemented in the system also offers 'partial-replacement' wherein we replace a frequently updated file after its minimum expiration period. Thus if a file is frequently updated it is overwritten with subsequent newer versions. Nevertheless incorporation of cache replacement policies including the traditional policies (Least Recently Used (LRU), Least Frequently Used (LFU)) and the recent replacement policies for Proxy Caching (LFU with Dynamic Aging (LFUDA), Greedy Dual-size Frequency (GDSF))[34] are worth investigating for peer-to-peer cache systems.

- System failure: We would like to investigate the point where the system might suffer drastically or fail so that we can improve the system and make it more robust. Currently we ran the system over multiple machines with their memory, hence the system is not bounded by cache space. Also, in the future more web content will be dynamically generated depending on user input and preferences and therefore not be suitable for any kind of caching. Our system works well with static web pages. An increased ratio of dynamic web contents would affect the cache consistency and performance.

[1]Subjective Logic is a logic which operates on subjective beliefs about the world and uses the term 'opinion' to denote the representation of subjective belief

Appendix A

Program code

hash.java module to obtain hashes of a string

```
// import package(s)
 import java.security.*;

// This class produces hashes of a string
 public class Hash {
   private final static int FILENAMELEN = 16;
   private final static int NODEIDLEN = 16;
   private final static int KEYLEN = 16;

//input:IP address and port number of the node
//output: the node string to be hashed with the format IPaddress:port number
 public static final String getNodeID(String address, int port)
    {
      String input = new String(address + ":" + String.valueOf(port));
      return getNodeID(input);
    }

//obtain the 16-byte node identifier
 public static final String getNodeID(String input)
    {
```

```
      String hashedStr = adjustLen(sha(input),Constant.NODEIDLEN);
      return hashedStr;
    }

//obtain the 16-byte URL identifier
 public static final String getFileName(String url)
    {
      String hashedStr = adjustLen(sha(url), Constant.FILENAMELEN);
      return hashedStr;
    }

//hash the string by using SHA-1 hash function
 private static String sha(String input)
    {
      String strDigest = new String();
      try
        {
          //obtain from SHA-1 hash function
          MessageDigest md = MessageDigest.getInstance("SHA-1");
          byte[] inputBytes = input.getBytes();
          md.update(inputBytes);
          byte[] digest = md.digest();
          strDigest = adjustToPrintable(digest);
        }
        catch (NoSuchAlgorithmException e)
        {
          System.out.println("In SHAhash (hash): " + e);
        }
        return strDigest;
    }

//take only the first 16 byes of the hash generated
 private static String adjustLen(String input, int len)
```

```
    {
      int sLen = input.length();
      String output = new String(input);
      if(sLen < len)
        {
            for(int i = 0; i < len - sLen; i++)
                output = output.concat("=");
        }
      else if(sLen > len)
        {
            output = output.substring(0, len);
        }
        return output;
    }

//obtain the string with all characters uppercase
 private static String adjustToPrintable(byte[] input)
    {
       char[] output = new char[input.length];
        for(int i = 0; i < input.length; i++)
        {
            if(input[i] > 0)
            {
              //perform the modulus division by 26 and add character 'A'
              output[i] = (char)(((int)input[i] % 26) + 65);
            }
            else
            {
              output[i] = (char)((((int)input[i]) * (-1) % 26) + 65);
            }
        }
        return new String(output);  }   }
```

Bibliography

[1] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *Proceedings of the 1996 USENIX Technical Conference, San Diego, CA*, pages 141–152, January 1996.

[2] R. Fielding. *Hypertext Transfer Protocol, version 1.1, RFC2616.* `ftp://ftp.rfc-editor.org/innotes/rfc2616.txt`, June 1999.

[3] D. Wessels and K. Claffy. ICP and the Squid Web Cache. *IEEE journal on selected areas in Communications*, 16(3):345–357, April 1998.

[4] J. Xu, J. Liu, B. Li, and X. Jia. Caching and Prefetching for web content distribution. *IEEE Computing in Science and Engineering*, August 2004.

[5] D. Wessels and K. Claffy. *Internet Cache Protocol (ICP), version 2, RFC2186.* `ftp://ftp.rfceditor.org/in-notes/rfc2186.txt`, Sept 1997.

[6] Keith W. Ross. Hash-Routing for Collections of Shared Web Caches. *IEEE Networks*, 11(6):37–44, Nov 1997.

[7] M. Hamilton, A. Rousskov, and D. Wessels. *Cache Digest Specification, version 5.* `http://www.squid-cache.org/CacheDigest/cache-digest-v5.txt`, December 1998.

[8] D. Wessels and K. Claffy. *Application of Internet Cache Protocol, version 2, RFC2187.* `ftp://ftp.rfc-editor.org/in-notes/rfc2187.txt`, Sept 1997.

[9] Vinod Valloppillil and Keith W. Ross. *Cache Array Routing Protocol, version 1.0.* `http://icp.ircache.net/carp.txt`, February 1998.

[10] R. Rivest. *The MD-5 Message Digest Algorithm.* `http://www.ietf.org/rfc/rfc1321.txt`, April 1992.

[11] S. Androutsellis-Theotokis and D. Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys*, 36(4):335–371, December 2004.

[12] M. Hefeeda. Peer-to-Peer Systems: A Comprehensive Survey. Technical report, Simon Fraser University, September 2004.

[13] M. Hofmann and L. R. Beaumont. *Content Networking - Architecture, Protocols and Practice.* Morgan Kaufmann Publishers, 2005.

[14] S. Ratnasamy, P. Francis, M. Hadley, and R. Karp. A scalable content-addressable network. *Proceedings of 2001 ACM SIGCOMM Conference*, March-April 2001.

[15] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of 2001 ACM SIGCOMM Conference*, 2001.

[16] A. Rowston and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *IFIP/ACM Middleware 2001*, November 2001.

[17] `http://www.napster.com`. *The Napster Homepage.*

[18] `http://gnutella.wego.com`. *The Gnutella Homepage.*

[19] `http://www.kazaa.com`. *The Kazaa Homepage.*

[20] H. Balakrishnan, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in P2P Systems. *Communications of the ACM*, 46(2), February 2003.

[21] M. Kaashoek. *Distributed Hash Tables : simplifying building robust Internet-scale applications.*

[22] D. A. Menasce. Scalable p2p search. *IEEE Internet Computing*, 7:83–87, March-April 2003.

[23] R. Flenner. *Java P2P Unleashed.* Sams Publishing, 2003.

[24] `http://spec.jxta.org/v1.0/docbook/JXTAProtocols.html\#proto-pdp`. *JXTA peer discovery protocol.*

[25] D. Eastlake and P. Jones. *US Secure Hash Algorithm 1 (SHA1).* Network Working Group, September 2001.

[26] S. Iyer, A. Rowstron, and P. Druschel. SQUIRREL: A decentralized,peer-to-peer web cache. *12th ACM Symposium on Principles of Distributed Computing (PODC 2002)*, July 2002.

[27] R. Malpani, J. Lorch, and D. Berger. Making World Wide Web Caching Servers Cooperate. In *Fourth International World Wide Web conference*, `http://www.w3.org/Conferences/WWW4/Papers/59`, December 11-14 1995.

[28] S. Ratnasamy, I. Stoica, and S. Shenker. Routing Algorithms for DHTs : Some Open Questions. *First International Workshop on Peer-to-Peer systems (IPTPS'02)*, LNCS 2429:45–52, March 7-8 2002.

[29] M. Ripeanu. Peer-to-Peer Architecture Case Study: Gnutella Network. *First International Conference on P2P Computing (P2P'01)*, 2001.

[30] L.A. Adamic and B.A. Huberman. Zipf's law and the Internet. *Glottometrics 3*, pages 143–150, 2002.

[31] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. On the implications of Zipf's law for Web caching. Technical report, University of Wisconsin, Madison, April 1998.

[32] X. Li, J. Liu, and M. R. Lyu. A trust model based routing protocol for secure ad hoc network. *IEEE Aerospace Conference*, 3, March 2004.

[33] A. Josang. A logic for uncertain probabilities. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 9(3):279–311, 2001.

[34] J. Dilley and M. Arlitt. Improving proxy Cache Performance: Analysis of Three Replacement Policies. *IEEE Internet Computing*, 3(6):44–50, November 1999.