Distributed Collaborative Caching for WWW

by

Wei Liu

A thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

Distributed Collaborative Caching for WWW

BY

Wei Liu

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of Master of Science

Abstract

The *World Wide Web* (*WWW*) offers an opportunity for users to share information and collaborate on activities globally. Better performance of the WWW is expected. This thesis presents a model and prototype implementation of a system that applies distributed system management techniques to the WWW. This system (called *Distributed Collaborative Caching for WWW*) provides a large distributed shared memory that is shared across a set of browsers in a network with no central server involved. This is a fully distributed approach for document sharing between browsers. The collaboration and sharing are supported by a shared virtual memory paradigm.

The major design contributions in this thesis are:

1. Greater availability of documents even if a remote server has crashed because a copy of the document may be accessed from the *Distributed Share Memory* (DSM). This helps to isolate end-users from server and network failures.
2. Decreased response times when retrieving documents because documents may be retrieved locally from the DSM.
3. Reduced server load and network traffic because requested documents can be delivered to the clients directly from DSM without a connection to the remote server.

This approach has some benefits beyond the popular proxy caching approaches:

- No proxy caching server.
- Use collaboration (between browsers) instead of centralization (proxy caching).
- Guarantee high scalability and reliability.
- No "single point of failure".

A prototype *Distributed Collaborative Caching for WWW* is implemented and its performance is measured and evaluated. The results are promising.

Acknowledgements

First and foremost, I would like to begin by thanking my co-supervisor, Dr. Randal Peters, for both his technical and moral support. I would also thank Dr. Randal Peters for providing me with a fellowship for most of my graduate studies.

I would like to thank my co-supervisor Dr. Ken Barker for taking the time to read this thesis. I would also thank Dr. Ken Barker for his constructive comments and suggestions, and his enthusiasm and encouragement.

I would like to thank my thesis committee member Dr. Bob McLeod for his time and remarks in preparing for the thesis defense.

I would also like to thank the other graduate students in the computer science department, whose companionship made graduate studies an entertaining time. Special thanks to Wanjie Wang and Raj Kumar Jayapalan for their helpful suggestions and comments on my implementation.

I extend my heartfelt thanks to my wife, Qing Wang, for her love, support, and faith in me.

Finally, I would like to thank my parents and other family members for their emotional support and encouragement all through.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Global information systems have become a reality with the advance of the Internet. Several applications have been developed (eg., gopher, WAIS, *etc.*) to access distributed information from anywhere in the world. The World Wide Web (WWW or Web) offers an opportunity for users to share information and collaborate on activities globally. This is enabled by interconnected networks and a body of WWW protocols and standards.

The WWW, which is based on client-server architecture, is a networking environment that consists of a set of distributed clients and servers. The servers in the WWW are the workstations that store, generate and transfer their documents to the clients that request them. The clients in the WWW are the browsers (eg., Netscape, Internet Explorer) that make requests, receive documents from servers and display the results. In fact, the WWW can be viewed as a navigational tool for information discovery and retrieval. Once information is available on the WWW, it should be accessible from any type of computer platform (Unix, MS Windows,

Macintosh, *etc.*) regardless of their location. WWW users use browsers to obtain information by retrieving web pages. These pages can contain graphics, video, and sound with text and include links to other pages located anywhere in the world. WWW pages are created using HyperText Markup Language (HTML) [HTM], which is a document "tagging" language that provides information links to reach other documents without requiring specific knowledge about the remote computer. HTML is also used to control document presentation within the browsers so all such browsers must understand HTML.

Behind every link in a page is the network-wide address of the page to which the link refers. This network-wide address is called a Uniform Resource Locator (URL). Every document on the WWW is uniquely addressable by the URL which contains the protocol, the server, and the document type. Once a document is published on the WWW, it can be located and transfered over the Internet using Hypertext Transfer Protocol (HTTP). The current WWW environment does not provide enough support for collaboration between browsers. Currently each request from a browser to a server requires two network transmission: one to send the request to the server and one for the server's response. These transmissions are subject to various network and server delays so any reduction in each traffic is bound to pay dividends. If an application running directly on the user's machine can facilitate information sharing among browsers, the network transmission will be reduced and the response time will be improved. This thesis designs and implements a system that provides a peer-to-peer collaboration between browsers which results in faster response time and fewer network transmissions.

1.1 Problem Area and Motivation

As the WWW becomes more popular, its continuing growth results in some significant problems. Performance, reliability and scalability are three major concerns.

1. Performance: Although the WWW enhances global accessibility, fast response times are still expected by users. Performance problems usually result from high network traffic and overloaded servers.

2. Reliability: Some collaborative applications such as flight information and stock prices require transparent failure recovery. Transient failures should not be noticed by users. Reliability problems are due to both server and network failure because it is impossible for a client to retrieve documents from a failed server or through failed network connection.

3. Scalability: The WWW is a kind of single server, multiple clients model. With the rapid increase number of users and information on the WWW, techniques must be developed to ensure that the servers' current ability to scale will not be compromised.

One approach that relieves these problems is *site mirroring*. This approach periodically replicates information of popular WWW sites to other sites known as "mirror sites". Mirroring is well suited for the replication of large, relatively static files that are frequently accessed by many users. However, mirroring lacks flexibility because users do not necessarily know where the mirror sites are and administrators

do not know which archives would be most profitably mirrored. Moreover, mirrors tend to become incomplete and dated.

Another approach to improve WWW access is caching. When a document is retrieved by a user, it is stored locally so that subsequent requests may retrieve the cached copy. This means a user does not need to know the location of replicated documents. Caching also relieves administrators from issues around replica management associated with the site mirroring approach because caching is demand-based and changes dynamically so it only caches popular documents.

Caching improves response times for accessing documents. Accessing a local copy of document is always faster and cheaper than accessing over a network connection. This approach also saves bandwidth by reducing the number of messages transmitted over the network. Studies, simulations and real-world experience have shown that WWW caching can significantly reduce network traffic and decrease latency [DHS93, Wor94, ASA+95, LAJF, AFAW97]. Most previous and ongoing research focuses on proxy caching servers, which are usually installed on the network path where multiple clients accessing the Internet can share the cache. Unfortunately, proxy caching still exhibits a centralized "nature", even if they are built as hierarchical structure like Squid [Squ]. Thus, a single proxy caching server will form a bottleneck so only a limited number of clients can use the same cache. Finally, as with any bottleneck this architecture also introduces a single point of failure.

All these existing issues help motivate this thesis. Can we make browsers caches collaborate with each other so there is no centralized proxy caching server? If this can be achieved, documents can be shared among browsers. Therefore, if one client has retrieved the document, then subsequent requests from other clients can get the

document that was already retrieved by the first client without traversing the links and connecting to the remote servers. This is a fully distributed approach compared to proxy caching approach so it has better scalability and reliability. This thesis present a model and prototype implementation of such a system. This system (called *Distributed Collaborative Caching for WWW*) provides a large distributed virtual memory that is shared across a set of browsers in a network with no proxy caching server involved. Collaboration and sharing among browsers are supported by a shared virtual memory paradigm.

1.2 Thesis Overview

WWW caching has proven to be an effective technique to improve the performance of the WWW. Proxy caches, like Squid, have been widely used around the world. Squid consists of a set of hierarchical proxy caches. Each proxy caching server is configured to interact with some subset of its peer servers as *parents, children* or *siblings*. It fetches the document from the first proxy that responds with the document. If the document cannot be found for the first query, the request will be forwarded up the hierarchy or multicasted to siblings. The query process is then repeated at the next level of the hierarchy. Unfortunately the centralized nature of the proxy caching servers causes scalability problems and a single point of failure due to a hierarchical structure. Several researchers have confirmed these observations [GCR97, FCA98, TDVK98].

The key idea behind the *Distributed Collaborative Caching* architecture is to make browser caches collaborate without a centralized proxy caching server. The

browser collaboration is supported by *Distributed Shared Memory* (DSM).

The highlights of this thesis work are:

- Design a *Distributed Collaborative Caching* model based on browser sharing for WWW. The collaborative cache environment, where browser caches are willing to cooperate to some degree to provide improved performance for all of their clients, is provided by DSM. Retrieved documents are stored in a shared memory and are shared among browsers through the distributed shared memory paradigm (Chapter 4).

- Implement a distributed collaborative caching prototype capable of sharing documents among browsers in a local network (an intranet) or across the Internet.

- Measure and evaluate the performance of distributed collaborative caching. The performance results are essential for designing, modeling, and tuning performance of WWW caching (Chapter 5).

The major design contributions in this thesis are:

1. Greater availability of documents even if a remote server has crashed because a copy of the document may be accessed from the DSM. This helps to isolate end-users from server and network failures.

2. Decreased response times when retrieving documents because documents may be retrieved locally from the DSM.

3. Reduced server load and network traffic because requested documents can be delivered to the clients directly from DSM without a connection to the remote server.

Although all of these contributions are provided by proxy caching to some degree, the approach proposed in this thesis provides some contributions beyond the proxy caching approaches:

- No proxy caching server.
- Use collaboration (between browsers) instead of centralization (proxy caching).
- Guarantee high scalability and reliability.
- No "single point of failure".

Chapter 4 will explain how our system achieves these advantages over the other approaches which are described in detail in Chapter 2.

The remainder of this thesis is organized as follows. Chapter 2 discusses the related work on improving document access on the WWW. It investigates and evaluates some caching techniques for the WWW. Chapter 3 gives the technical background and existing technologies that motivate our design. Chapter 4 presents the general model architecture and details the design and implementation of the distributed collaborative caching for WWW. Chapter 5 describes the experiments and shows experimental results. Finally, conclusions and future work are presented in Chapter 6.

Chapter 2

Related Work

While many WWW caching systems have investigated performance improvement on the WWW, no definitive solution has yet been reported so new solutions continue to be sought. One such proposal is described in this thesis.

The fundamental concept of WWW caching is to store the retrieved documents from remote servers close to the clients that request them. If the same document is requested a short time later, it can be found in the cache so no connection to the remote server is necessary. WWW caches are typically implemented using three approaches: *client (browser) cache*, *server cache*, and *proxy cache*.

Browser caches are local to the machine running the browser that is interacting with the user. They are the first level of caching and not shared between users in the current WWW environment. Our approach makes browser caches collaborate to share their cached documents. Server caches are implemented to reduce the origin server load by duplicating their documents at a second caching machine. Caching servers usually sit side by side with the origin server to service requests.

If none of the cache servers contains the requested document, then the request is passed to the origin server. Proxy caches are installed on a firewall proxy server. These proxy servers are usually located on the network path from multiple clients to multiple servers. Normally, a proxy server acts as both a client and a server. It is a server when accepting HTTP requests from clients connecting to it, but it acts like a client when it connects to the remote server to retrieve the documents for its own clients.

This chapter evaluates several WWW cache systems to assess the performance improvement of the WWW.

2.1 The CERN Proxy Cache

The CERN cache is the original WWW proxy and has set a number of standards for proxying and caching on the WWW. The CERN server [LA94] was designed for people to access to the WWW within a firewall as shown in Figure 2.1.

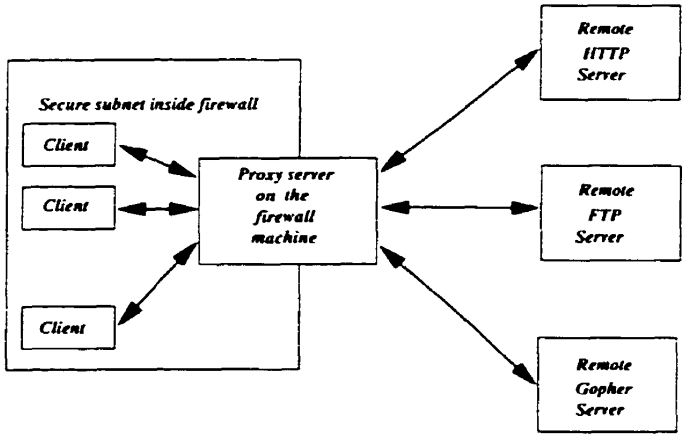

Figure 2.1: A Proxy on a Firewall Machine

The CERN proxy can function as a proxy server and as a cache. The designers believe that caching is more effective on the proxy server than on each client. As a proxy cache, the operation of a CERN proxy is illustrated in Figure 2.2.

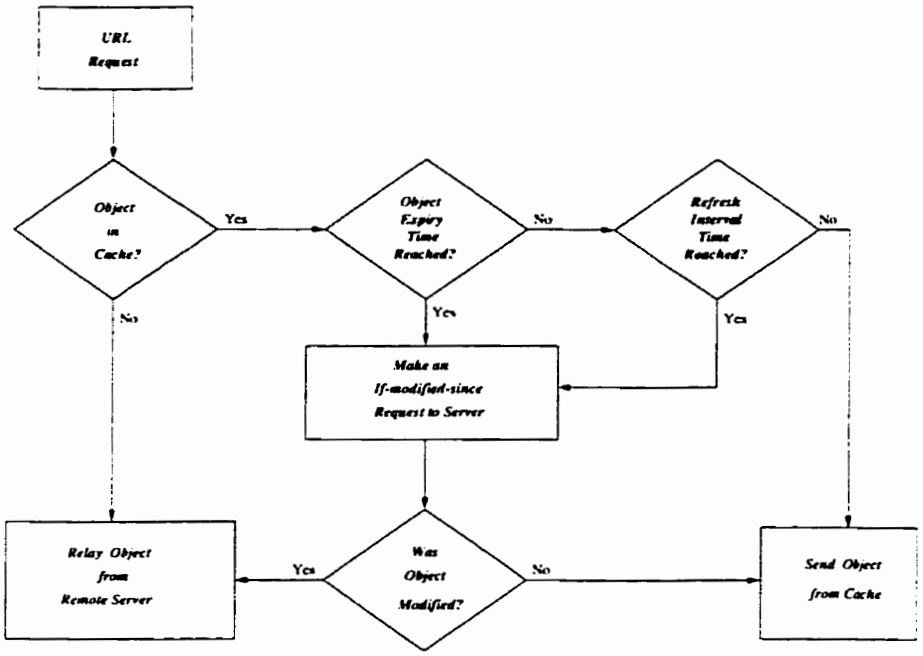

Figure 2.2: CERN Proxy Flow Diagram

The CERN proxy forks a new process to handle each request. It caches both data (WWW documents) and proxy meta-data (expiration time, document type, *etc.*) using the underlying file system. A request is translated into a file name, which is a path consisting of URL components. The URL directory is the path without the last component. The meta-data is stored in a separate file for each URL directory. When the proxy server receives a request, CERN tries to open the meta-data file and every component of the URL directory should be matched. If the meta-data exists and it shows that the cached document has not expired, the document is sent back from the cache. If the meta-data is not present, CERN

forwards the request to the remote server and passes the reply to the client and stores it in the cache.

Cached documents are returned to the client without checking the remote server until the document expires in the cache. If the cached document expires, an "if-modified-since" request is issued, which returns the document only if it has been modified since the last retrieval.

The CERN proxy architecture is simple and usually deployed as a non-hierarchical flat structure on the client-side. However, this is a centralized approach that does not scale well when the number of browsers wishing to share documents increases. Our approach distributes the management of documents across the participating browsers and hence provides good scalability as the shared environment grows.

2.2 Squid Hierarchical Cache

The Squid proxy [Squ] is designed to improve the performance of the CERN cache. It is the next-generation of the Harvest cache [CDN+96].

Squid is a set of hierarchical proxy caches. It is motivated by the need to use hierarchical caching on the Internet by challenging the conventional belief that hierarchical caching does not merit the cost. Squid arranges caching proxy servers into a hierarchy from stub networks to regional networks to backbone networks (Figure 2.3).

Each proxy cache can interact with other servers which are its *parents*, *children*, or *siblings* by multicasting a query. Squid uses logical multicast which repeats the unicast message to all the recipients. The requesting server fetches the document

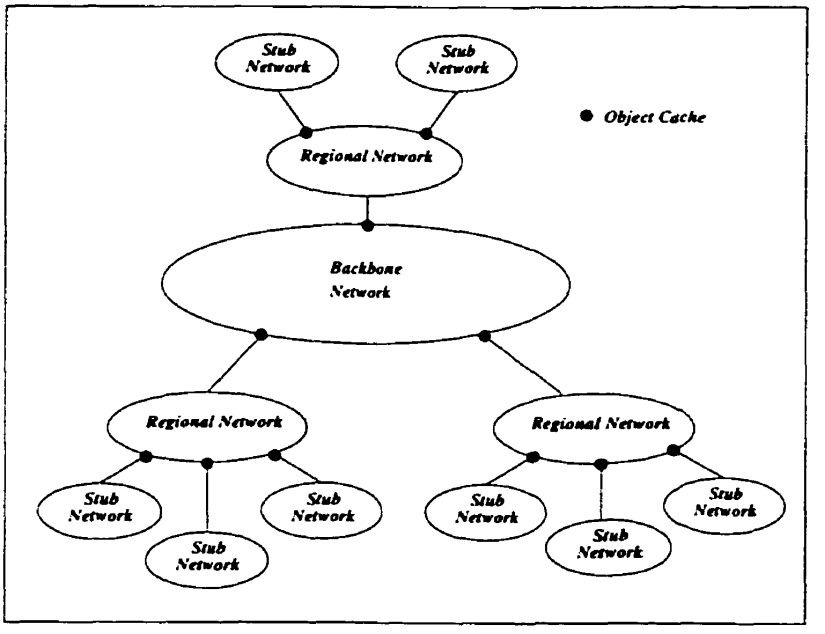

Figure 2.3: Hierarchical Cache Arrangement

from the first proxy that responds with a cache hit[1]. If there is no hit, the requesting server fetches the document through the first parent that responds with a cache miss[2] instead of going to the remote server directly. If the requesting server has no parents, or no parents respond within a timeout interval, it fetches the document from the remote server. Each cache along the path caches the fetched document locally.

The Squid cache uses non-blocking network I/O and avoids forking new processes except for relaying FTP requests, which are retrieved via an external process. To avoid another potential source of blocking, Squid also includes a customized DNS server and cache. In managing its own resources, Squid attempts to isolate itself

[1] A cache hit is a client request that is successfully satisfied without transmitting the document from the remote server.

[2] A cache miss is a client request that is not a hit.

from the operating system. It keeps meta-data (URL, TTL, reference counts, disk file reference and various flags) for cached documents in virtual memory. This enables Squid cache to determine whether it can serve a given request from its cache without accessing the disk. Documents in the cache are referenced via a hash table keyed by URL.

The goals of the hierarchical Squid cache are to distribute load away from server "hot spots" and to reduce access latency, provide better network bandwidth usage, and balance server loads. When appropriately setup and tuned hierarchical caching can be effective. However, the static nature of the hierarchy does not adjust well to changing document traffic so additional tuning is needed to maintain performance. Moreover, a considerable amount of cooperation is required to setup a hierarchy and this becomes increasingly difficult. Management of information about siblings is becoming more and more complex as the responsibility for maintenance is disseminated throughout various organizations within a caching network. In addition, hierarchical caching requires large disk space at upper level caches. As the number of users increase, the root cache and upper level caches will have to increase their storage capacity proportionally to maintain efficiency. Although Squid caches are arranged in a hierarchy, the centralized nature of proxy caching servers will limit scalability and result in a single point of failure problems. Clearly a distributed approach to manage cached documents that does not use any central proxy servers will avoid single point of failure problems and provide better scalability and reliability.

2.3 CRISP Distributed Cache

A CRISP cache is a directory-based distributed caching proxy that consists of several cooperating caching servers sharing a centralized global directory.

In CRISP cache, a central mapping server is used to keep track of the cache contents of all participating proxies. Each client is bound to one of the proxies, which cache documents for their clients (Figure 2.4).

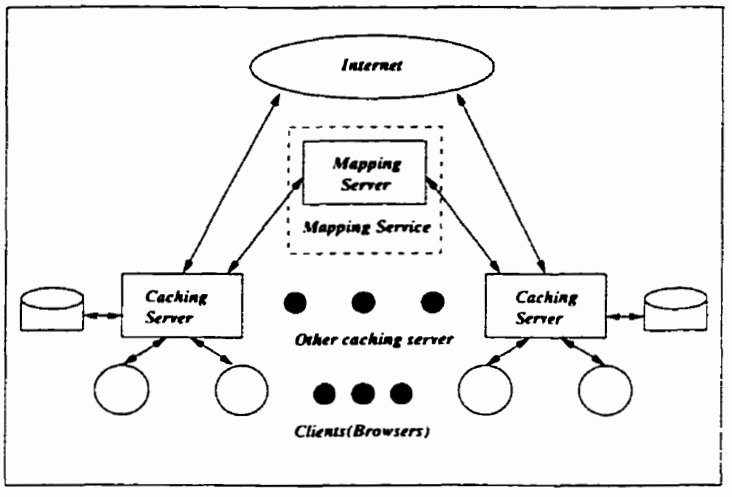

Figure 2.4: A CRISP distributed Cache

The CRISP cache is cooperative: when a browser requests a document through a caching proxy server, the proxy forwards the request to the central mapping server which maintains a complete cache directory. If the requested document is cached in some other proxy, then the mapping server instructs the proxy holding the document to send it to the requesting server without accessing the remote server. If the requested document does not exist at any proxy, then the requesting server retrieves the document from the remote WWW server. The proxies inform the mapping server whenever they add or remove a document from the cache.

CRISP's central mapping service is simple. Proxies probe the entire cache with a single unicast message exchange. The authors argue that multicasting increases network traffic and forces all caches to respond to each request in most cases. In addition, a CRISP proxy server does not need to query its siblings (unlike Squid) so it avoids maintaining sibling information.

If the mapping server fails, the proxy can still service its clients by retrieving documents directly from remote servers until the map functions again. This does not interrupt service but only degrades performance to the worst case scenario. If a proxy fails, the mapping server will mark all cache directory entries from that cache as unavailable. When the failed proxy rejoins, it first registers itself with the mapping server to re-enable or restore the maps for cached documents that survived the failure.

CRISP supports distributed caches by registering more proxies with the mapping server and requests can be redirected to any participating proxy. A global Least Recently Used (LRU) scheme can be used to simplify load balancing and cache consistency can be implemented by using the mapping server as a single point of entry for invalidation.

CRISP relies on the central map that keeps track of the location of all cached documents. This prototype is referred to as CRISP/CSD (Central Synchronous Directory). The drawback of this approach is that the central server can become a performance bottleneck. This is because once a document is cached at a proxy, requests from both other caches and its own clients are sent to it. This can also create hot spots.

Three alternatives, *Partitioned Directory* (PD), *Replicated Directory* (RD) and

Replicated Partial Directory (RPD), are described by Sadde *et al.* [GCR97] to improve scalability, reduce latency, and improve resilience to mapping server failure. Some similar strategies for the placement of directory in a Distributed Shared Virtual Memory (DSVM) system are presented by Mathew *et al.* [MGB95, MGB96].

A PD distributes the map among the caching servers. The mapping service consists of several mapping servers and they do not communicate with each other. All participating caches only send queries and updates to the assigned mapping server. A partitioned directory can provide some benefits. It can be scalable to large numbers of caching servers because map query load is distributed among a number of map sites. However, there are some disadvantages in this strategy. The reliability of the system is not improved greatly because any failure of a mapping server render some objects inaccessible.

A RD replicates the global directory at every caching server. Requests for directory services can be serviced locally. The likelihood of bottleneck is reduced and the reliability of the system is enhanced. No single proxy caching server can become the single point of failure in the system as long as a request for a directory service can be routed to another replica. However, Mathew *et al.* [MGB95, MGB96] argue that a full replication is impractical because preserving directory consistency is extremely costly. If the directory is replicated at all nodes, each modification of the directory would require a lock request to be broadcast to all nodes. This would result in an excessive amount of network traffic. An approach for alleviating these problems is to propagate the information to other replicas in a lazy fashion. As proposed in CRISP, updates to the global map are propagated asynchronously. Another disadvantage is that the storage requirements to maintain a directory replica

are as expensive as with a centralized directory. If the size of a directory becomes cumbersome, a directory compression scheme such as *Summary Cache* [FCA98] can be used.

A RPD is considered to be the best approach among these alternatives. In RPD, the map is distributed among the caching servers and only the most valuable portion of the map (submap) is replicated at each mapping server. Submap replicas can be updated lazily, which reduces the storage and update cost. A RPD combines the advantages of both centralize directory and partitioned directory. Replication of the directory fragments enhances the reliability of the system. Even if a caching server with a submap fails, the mapping server with a replica of the submap can still provide services. One disadvantage of this strategy is that the consistency of the directory is difficult to maintain because of replication of directory fragments. Another disadvantage is that registration of an object requires the nodes with replicas of a fragment to communicate with each other.

2.4 Summary Cache

Summary cache [FCA98] is a scalable wide-area web cache sharing protocol. Each summary is stored as a "Bloom filter" [Blo70] which is an efficient hash-based probalistic scheme. It is used to represent a cache directory within a summary cache.

A Bloom filter is a method for representing a set $A = a_1, a_2, ..., a_n$ of n keys to support membership queries. Within a summary cache, a proxy builds a Bloom filter from the list of URLs of cached documents and sends the bit array plus

the specification of the hash functions used to other proxies. When updating the summary, the proxy specifies which bits in the bit array are flipped. Each proxy maintains a local copy of the Bloom filter and updates it as documents are added to and replaced from the cache. To update the local Bloom filter, a proxy maintains an array of counters, each counter remembering the number of times the corresponding bit is set to 1. When a document is added into the cache, the counters for the corresponding bits are incremented. When it is deleted from the cache the counters are decremented. When a counter increases from 0 to 1 or drops from 1 to 0, a record is added to a list which will be sent to other proxies at the next summary update.

In this protocol each proxy keeps a summary of the cache directory of each participating proxy and checks these summaries for potential hits before sending queries. When a miss occurs, a proxy first checks all summaries to find if the requested document is in other proxies. If a summary shows a potential result, then a query is sent only to that proxy to fetch the document. If it is not, the request is sent to the WWW server directly. When documents are added to or replaced from the cache the local copy of the Bloom filter is updated.

There are two issues that affect the scalability of summary cache: summary update and summary size. Frequent summary updates will incur a lot of network overhead. In the proposed protocol, the summary is updated lazily. A proxy does not update the copy of its summary stored with other proxies upon every modification of its directory. It only broadcasts the summary changes to all other proxies whenever the percentage of its cached documents are not reflected in other proxies. For performance reason, the summaries are stored in the main memory

because of the faster memory lookups. Individual summary size and the number of the cooperating proxies will determine the memory requirement. Keeping the summaries small is very important because the memory grows linearly with the number of proxies.

2.5 WWW-DSVM

WWW-DSVM allows sharing of retrieved documents among browsers through an underlying shared virtual memory paradigm. In the operating system context, a *Distributed Shared Virtual Memory* (DSVM) [BPG95, GB93, GBBZ92] provides a single, common virtual address space, which is shared by multiple objects distributed across a network. In DSVM, data sharing is easy from the application perspective since all data appears to all processes at the same virtual address. Moreover, the distribution of objects is transparent and objects can be relocated between sites while maintaining their shared virtual address.

Based on these considerations, WWW-DSVM [Sar97, SPB98] presents a very different approach to WWW caching. This approach uses a DSVM to provide a distributed cache among browsers in a network for sharing documents. Retrieved WWW documents are stored in the DSVM and subsequent requests for the same document retrieves them from the shared memory without connection to the remote server.

The DSVM is a transparent layer between the WWW servers and the browsers. It is managed by the *Distributed Shared Memory Manager* (DSM-M), which provides a single virtual memory abstraction among processes at all nodes and main-

tains shared memory consistency across sites. The WWW-DSVM system uses Treadmarks [KAZ+96] as the DSM-M to provide shared memory across sites.

The WWW-DSVM is composed of a Domain O (Domain outside of the DSM-M) and a Domain I (Domain controlled by the DSM-M) as shown in Figure 2.5.

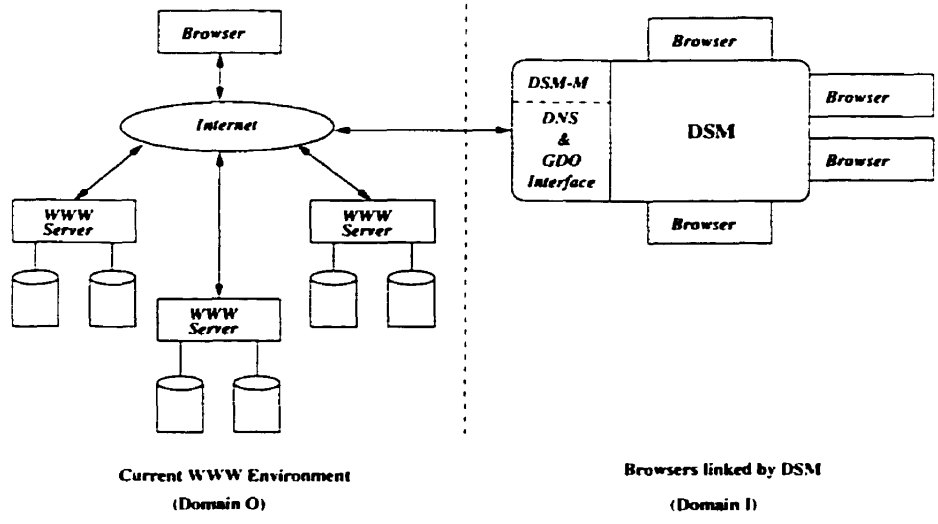

Figure 2.5: WWW in DSVM

Domain O is the current WWW environment. Domain I is the architecture based on the DSVM approach. The DSM-M has two main components: the Domain Name Service (DNS) and the Global Directory of Objects (GDO). The GDO is used to manage the shared documents of WWW in DSVM. Each browser manages its own address space and has its own local cache. DNS and GDO provide the interface between the DSVM and the WWW. The interface uses the DNS and standard protocols (HTTP, FTP and Gopher) for retrieving documents from the WWW.

In this prototype, browsers are linked to the DSVM through a set of proxy servers that wait for requests from browsers. These proxy servers are used to

perform the duties of the DNS and GDO interface to provide interoperability for multiple browsers. The participating browser machines are set up as Treadmarks servers to manage the documents in shared memory.

The retrieved documents are placed in the DSVM and are visible to all browsers in DSVM. When a browser requests a document, it sends its request to a proxy server. The proxy server invokes a thread and sends the request to any of the Treadmarks server. The selected Treadmarks server checks the GDO in shared memory with the DSM-M if the requested URL exists. If an entry corresponding to the requested URL is found in the GDO and it is still valid, then the document is returned to the client from the shared memory. If the document in DSVM is outdated and the server is unreachable, the document is returned with a warning that it is a cached copy. If a requested document does not exist in the shared memory, then the thread requests the DNS and GDO interface to retrieve the document from the WWW server and the GDO is updated with the document details. The consistency in DSVM is controlled by using validation checks of the Apache Server [Apa]. Documents are refreshed when the copy in the DSVM is outdated.

WWW-DSVM provides sharing of retrieved documents among browsers through a shared virtual memory. A set of proxy servers function as an interface between browsers and DSVM. Document requests from all participating browsers will be sent to a proxy server first to check if the requested document exists in shared memory. However, this is still a centralized approach and has the same problems such as a single point of failure and bottleneck.

Chapter 3

Technical Background

This chapter details some technological achievements that motivate this research. These technologies include:

- World Wide Web
- WWW Caches
- Distributed Shared Memory
- Treadmarks

These core technologies form the basis for our system which provides improved WWW performance. The *Distributed Shared Memory* (DSM) system provides a distributed cache among browsers in a network for sharing documents. The distributed shared memory is provided and managed by Treadmarks. Shared memory consistency is accomplished with Treadmarks functions. Different WWW cache placement strategies and some WWW caching issues are discussed because they can affect the WWW caching design strategies and performance.

3.1 World Wide Web

The World Wide Web (WWW or Web) was first developed as a tool for collaboration at CERN [LA94]. From there it spread rapidly to other field and grew to its current size. There are extensive research efforts to provide more efficient access to WWW documents while reducing network traffic and server load. This research presents an approach that makes a set of distributed, heterogeneous browsers collaborate with one another to form a co-operative document sharing environment using a shared memory paradigm.

The WWW is one of many Internet-based communication systems consisting of all pages on the Internet. These pages are accessible anywhere in the world through a set of information protocols, standards, and conventions. These information protocols allow all the clients and servers to communicate. Some of the most common WWW protocols and utilities are 1) Resource Addressing: URL, 2) Data Transfer: HTTP, and 3) HyperText Markup Language (HTML).

Uniform Resource Locator (URL) is a standard for identifying objects on the Internet accessible via the WWW. A URL is a string of characters that uniquely identifies an object on the network. A URL specifies the address for any object anywhere on the Internet, though you access these objects using a variety of different protocols.

A URL has three basic sections as illustrated in Figure 3.1.

- Protocol - Determines which protocol the browser uses to contact the server.
- Server identification - Identifies the server to contact (can include options).

Figure 3.1: Sections of a URL

- File identification - Identifies the location and name of the document requested (can include a directory path and an optional section name).

The protocol portion of the URL is used by the browser system to communicate with the server. The protocols supported by browsers and servers, in most cases, are the standard protocols: *http* (for documents on a remote server) and *file* (for local documents). The server ID portion of the URL identifies the server where the information you want resides. The file ID in a URL specifies the directory, subdirectory (if any), and document name.

WWW browsers and servers communicate with each other using the HyperText Transfer Protocol (HTTP) [HTT]. The HTTP is an application level protocol for distributed, collaborative, hypermedia information systems. It builds on the reference provided by the Uniform Resource Identifier (URI), as a location (URL), or name (URN), to indicate the resource on which a method is to be applied. HTTP is also used as a generic protocol for communication between browsers and proxies/gateways to other Internet protocols. On the Internet, HTTP communication typically runs over the TCP/IP protocol suite using the default TCP port of 80 (although any valid TCP port can be used). HTTP only assumes reliable transport so it can be used with any type of transport/network protocols.

HyperText Markup Language (HTML) [HTM] is the standard for writing and formatting HyperText (also called Hypermedia) pages on the WWW. HTML is a

document coding language that allows structured text with links. HTML defines the logical structure of the document instead of its formatting. This allows browsers to display HTML pages on different platforms using different fonts and conventions. With HTML, you can create links to other documents, and embed graphics, sound, animation, or anything else a computer system can interpret.

3.2 WWW Caches

WWW caching has proven to be an effective technique for reducing latency on the WWW [ASA+95, LAJF, Woo96, TDVK98]. Different WWW caching design strategies will affect the performance of the WWW. This section presents some details on cache locations and discusses some cache issues that should be considered when design a WWW cache system.

3.2.1 WWW Cache Location

There are three logical locations for WWW caches. The proxy caching alternation is the main concern of this thesis. A brief description of the other two; client caching and server caching is also included in this section.

- Client Caches (Browser Caches)

  All popular WWW browsers available today have a built-in cache. There are two types of client caches: *persistent* and *non-persistent* caching. A persistent client cache retains its documents between invocations of the browser, while a non-persistent client cache deallocates any memory or disk used for caching

when the user quits the browser. Client cache sizes are usually in the range of 5–50MB.

Client caches make sense when a user accesses the same documents repeatedly. If the same document can be found in the cache and is valid, no connection to the remote server is necessary. If the cache does not have the requested document or the document is outdated then the request is passed to the remote server.

Client caches are typically not shared between users because they are local to the machine running the browser that is interacting with the user. Therefore, client caches are not very efficient with respect to the broader community.

- Server Caches

  Server caches are implemented to reduce the server load by duplicating their documents at a second caching servers. Unlike proxy caches or mirror sites, caching servers are usually installed side by side with the origin server to serve the requests. This essentially provides a parallelization at server services.

  Server caches are mainly used to improve the performance of the origin server. A request can be served by one of the caching servers. If none of them has the requested document then the request is forwarded to the origin server.

- Proxy Caches

  A proxy cache is an application-level network service for caching WWW objects. Proxy caches are usually installed on the network path where multiple clients are accessing the Internet. Therefore, proxy caches can be simultaneously accessed and shared by many users.

The primary use of proxies is to allow internal clients access to external servers within a firewall and are not necessarily used to cache the replies passing through them. The proxy application acts as an intermediary between WWW clients and the rest of the Internet.

Figure 3.2 presents a normal HTTP transaction that does not use proxies. In this situation, clients have their own IP address and connection to servers on the Internet. When a normal HTTP request is made by the browser, the HTTP server gets only the path and keyword portion of the requested URL. Other parts of the URL, such as the protocol specifier "http:" and the host name, are known by the remote HTTP server.

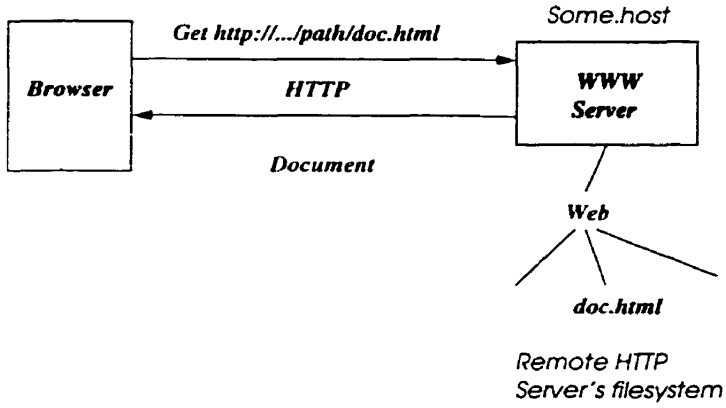

Figure 3.2: A Normal HTTP Transaction

When a user enters:

http://www.cs.umanitoba.ca/~user/information/thesis.html

The browser converts it to:

GET /~user/information/thesis.html

The browser connects to the server running on www.cs.umanitoba.ca, issues the command, and waits for a response. In this example, the browser makes a request to the HTTP server and specifies the requested resource relative to that server so there is no protocol nor host name specifier in the URL. The request specified the path (Data Directory) of information and the thesis.html document located in the Data Directory. The response is a document or an error message.

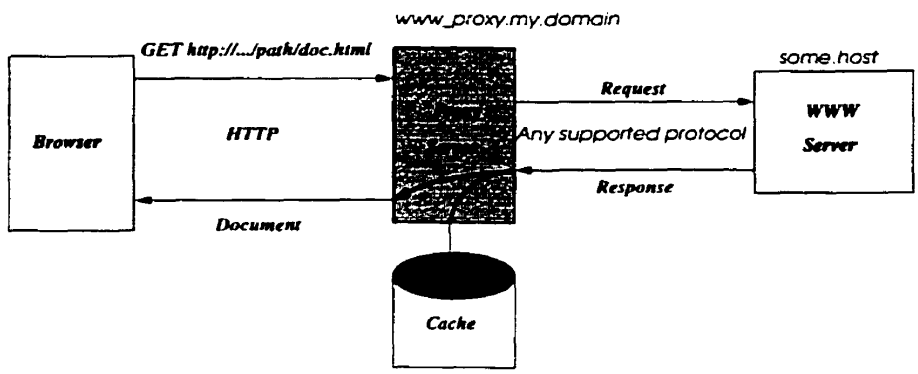

Figure 3.3: A Caching Proxy

Figure 3.3 shows a caching proxy. The proxy server acts as both a server system and a client system. It is a server when accepting HTTP requests from browsers and acts as a client system when its browser software connects to remote servers to retrieve documents. When a browser sends a request through a proxy server, the browser always uses HTTP for the transactions with the proxy server. This is true even when the user wants to access a remote server that uses another protocol such as FTP. Instead of specifying only the pathname and search keywords to the proxy server, the browser specifies the full URL. This way the proxy server has all the information

necessary to make the actual request to the remote server specified in the request URL using the protocol specified in the URL.

The clients inside the firewall make their requests to the proxy. The proxy forwards the requests to the remote server and then relays the replies back to the clients. The retrieved documents are then stored in both the client caches and the proxy cache. Since a proxy is usually used by all the clients on the same subnet and it relays traffic between clients and servers, it makes sense for the proxy to cache documents that are retrieved by several clients. If people with similar interests access the same documents or a user tends to browse back and forth between documents, these requests may be served to the client from the proxy cache without connecting to the remote server if the requested documents are stale in client cache. This is true because the same documents may be retrieved by other clients thus the proxy cache always has the latest documents than client caches have. This saves external network bandwidth and reduces the retrieval latency to the users.

A single proxy cache also introduces some problems such as scalability and robustness because a single proxy cache is both a bottleneck and a single point of failure. Having multiple independent proxy caches cooperate to service a group of clients can relieve some of these problems.

The purpose of any WWW cache architecture is to serve as many of the requested documents from the cache as possible. Thus, the most effective caches will be those serving the largest user communities. Therefore, making a group of proxy caches work together to cooperate can serve more users over a wide area than just a single proxy cache. The basic idea in cooperative caching is: if a requested

document is cached at any proxy in the set of cooperative proxy caches, the document is sent back to the user from that cache directly without accessing the remote server. Each proxy cache makes independent decisions to determine how it serves its users and all proxy caches should function equivalently to handle the request of any clients. However, this centralized approach has the same limitations as a single proxy server with a single point of failure and scalability problems.

Our approach takes this concept further. It allows the sharing of the documents between clients without proxy servers. Browser to browser collaboration is provided by DSM.

3.2.2 WWW Cache Issues

This section discusses which protocols to cache, caching consistency mechanisms, and some related issues. These issues may affect the caching design strategies, performance, and optimization of WWW caching.

1. **The Protocols**

   A proxy is a standard method for setting up a firewall. A client need only understand HTTP because other protocols can be handled by the proxy which speaks all the WWW protocols. Thus, a client makes a request to the proxy server using HTTP, the proxy server connects to the remote server and requests the document relative to that server using the actual protocol, and the result is sent back to the client using HTTP. By using HTTP between the client and proxy servers protocol functionality is not lost because FTP, Gopher, and other WWW protocols map to the HTTP method [LA94]. In

general, it is possible to cache documents retrieved using the HTTP, FTP and Gopher protocols.

2. **Caching Consistency**

   A main problem for caching is that some cached documents are stale when the remote server's copy is modified. Maintaining cache consistency is a hard problem because HTTP servers have no way to inform caches about updated documents. Currently, there are three cache consistency mechanisms in use on the Internet: *time-to-live fields*, *invalidation protocols*, and *client polling*. Each of these is described below.

   Time-To-Live Fields

   Time-To-Live (TTL) fields are an estimate of a document's lifetime used to determine how long cached documents remain valid. When the proxy is responsible for maintaining cache consistency a time-to-live value must be assigned to its documents. TTLs are usually implemented using an "Expires" header field or a "last-modified" header field in HTTP but FTP and Gopher do not include this field.

   If a cached document has not expired in the proxy server, the cached copy is returned to the client. If the "Expires" field is absent, then a "get-if-modified" request is sent instead of a "get". A "last-modified" value of the cached copy is sent within the request. If the header data returned with the document does not include a "last-modified" field then the cache server will use a default TTL value [CDN+96].

   A global TTL is hard to set because the lifetime of different types of documents or different domains varies greatly. Besides, a fixed TTL is not appro-

priate for some situations such as a dynamically generated document.

As the value of the TTL increases, the probability of returning stale documents does too. On the other hand, if the TTL value is too small the cache will have fewer valid documents and thus result in a lower hit ratio.

Worrell [Wor94] shows that a single TTL applied to all documents works poorly and the "Expires" field is rarely used with only six out of 28,000 documents contain this field.

Invalidation Protocols

Invalidation protocols depend on the primary server keeping track of cached documents on the proxy server. Each time a document changes the proxy caches receive a callback invalidation that their copies are no longer valid. Therefore, the responsibility lies on the primary server to notify all the proxies of updates.

Invalidation protocols are usually used when strict consistency is required. However, invalidation protocols are difficult to deploy because it is necessary to modify the server. Another problem is scalability because the primary server must maintain information about where their documents are currently cached. Invalidation protocols will also result in burst of synchronization traffic.

Handling machine failures is expensive. If a machine with documents cached cannot be notified, the server must continue attempting to reach it because the cache will not know to invalidate the document unless it is notified by the server.

Invalidation protocols can be used to improve cache consistency but the

proxy's standard consistency controls can be used as a backup if they fail.

Client Polling

Client polling (or validation checks) is a technique where clients check back with the primary remote server to determine if cached documents are still valid. When documents are retrieved from the remote servers the proxy server stores a timestamp along with it. When the cached document is requested, the proxy server checks with the primary server to determine if they have the same timestamp. If the stored timestamp differs from the one in the primary server a new version is sent back from the remote server.

Client polling has been implemented in HTTP. A "if-modified-since" header field indicates that the server should only return the requested documents if the documents have changed since the specified date. It is possible for the cache to return stale documents if the documents change during the time when the cached copy is considered valid. Thus the client polling does not support perfect consistency. Another problem is that the proxy cache will invalidate documents that are still valid which raises a performance issue. A proxy cache could invalidate at an interval time and wait until it is requested by the client. Although this might improve short-term performance, it will increase the network traffic.

Maintaining cache consistency in the WWW need not be expensive. Using only invalidation protocols may not be a good solution because of the scalability and the machine failure problems. Combining invalidation callbacks from the primary server with TTLs may be more useful. This is because if the TTLs expire and there has been no invalidation callbacks, or the primary

server is unable to contact the primary, the cache is alerted that a possible stale document is returning to the client.

This thesis uses Time-To-Live fields provided by WWW server to validate a document in the shared memory.

3. **Other Issues**

There are some other issues related to web caching such as uncachable documents, time critical data, paid-for documents, and document size that may affect the performance of the WWW. This section gives A brief description on these issues.

Not all types of documents on the WWW can be cached. With the evolution of the WWW, more and more documents are created dynamically and their contents are modified each time they are retrieved. For example, CGI (Common Gateway Interface) documents are generated dynamically from user queries and every reply may be different. Thus, user query results and any URLs containing *cgi-bin* are not cached.

Some documents are time dependent and the information providers want to ensure that users are never shown stale versions of their documents, such as weather, stock prices, and other time-sensitive documents.

Another concern is that the origin server cannot keep a precise log of accesses to their resources. If the information providers charge for access to their resources, caching of such paid-for documents will be unacceptable to them. One solution to this problem is for a proxy to maintain the number of hits and return this to the origin server so appropriate billing statistics can be maintained.

It is also possible to limit the size of the largest cachable document so that a few large documents do not purge a lot of small documents from the cache.

These issues will affect the performance of the web caching thus should be considered when design and develop web caching.

3.3 Distributed Shared Memory

Distributed Shared Memory (DSM) is an abstraction that supports the shared memory model in software of a distributed memory hardware. Figure 3.4 shows the conceptual representation of a distributed shared memory system.

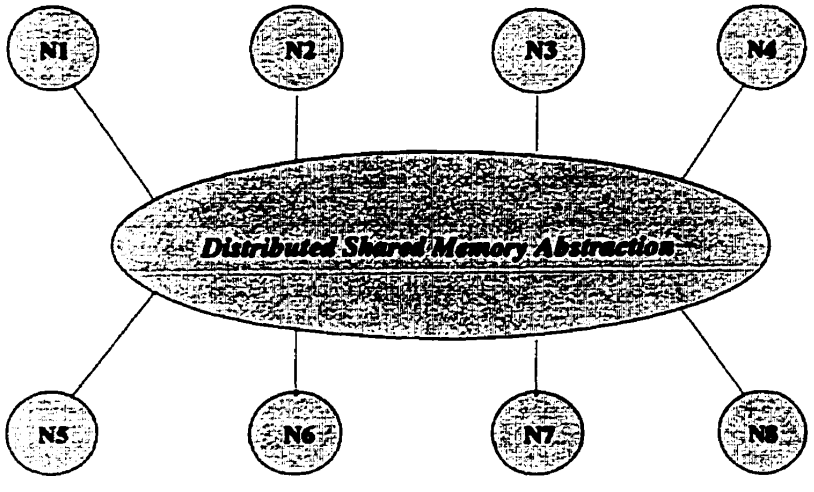

Figure 3.4: Distributed Shared Memory Abstraction

Such architectures contain no physical shared memory so they do not provide a physical global address space. Indeed the distributed local memories collectively provide a virtual address space shared by all processors connected via an interconnection network. In a distributed shared memory multiprocessor environment,

each processor can read or write any element in a common memory. Each processor has a large cache to reduce the amount of traffic between processors and common memory. Shared memory programming is considered easier because there is only one common memory so there is no need for the programmers to move data to make it accessible to another processor. Shared memory requires only that the programmers consider synchronization.

3.3.1 Shared Virtual Address Space (SVAS)

A DSM system provides a *Shared Virtual Address Space* (SVAS) [CLF94, OS92] shared by all nodes. The SVAS paradigm has a single, common, virtual address space that provides ease of programming by eliminating coding of cumbersome communication procedure between applications.

To support a global shared virtual address space, a DSM system treats the local memory of each node as a coherent cache and implements a cache coherency protocol. The system maps the shared address space at the same address on all system nodes. Any processes can refer to any address within the shared address space and any data item at a virtual address appears at the same address in all processes.

An SVAS system can be particularly beneficial in object-oriented systems. The distribution of objects is transparent and objects can move freely among sites. Access to a non-resident object will obtain the object from the site that holds the object. Objects need to be referenced using a unique, systemwide, and immutable *Object IDentifier* (OID).

Because the shared address space is never destroyed and is valid across all pro-

cesses and subsequent executions of these processes, the SVAS system can provide persistence for an object's state. When a process terminates, the object's pages can simply be swapped out to stable storage until some process needs the objects again. At this point, the object can be paged into memory as needed.

3.3.2 Distributed Shared Virtual Memory (DSVM)

A *Distributed Shared Virtual Memory* (DSVM) [BPG95, GB93, GBBZ92] system extends the SVAS concept to processes scattered across the nodes in a distributed system. It provides a persistent object-based DSM for the persistent management of shared objects. DSVM implements a DSM between interconnected nodes (workstations) of a network and these nodes are clustered into cells consisting of several nodes per cell [MGB96]. With the same address space visible to all processes on all nodes, each node's physical memory behaves as a local cache and each node's persistent swap or storage area can be linked to a processor's local memory in a multiprocessor in a DSVM system. Because objects in a DSVM system can be distributed across several nodes, a key data structure necessary for managing the objects in a DSVM system is an *object directory.* The directory maintains information a client node would need to locate an object in persistent storage. Additionally, the directory can also associate with an object such information as access rights to the object and concurrency control information. The directory could then provide services that would assist the DSVM in managing objects (eg., concurrency control, access control, location of objects in persistent storage, *etc.*).

A DSVM system simplifies distributed applications. Instead of requiring that the applications perform memory management and object storage operations them-

selves, DSVM could be called to perform these services for the applications. Applications could then simply reference objects into or out of memory. Thus, a DSVM system would be responsible for providing services to the kernel such as locating the pages that make up an object, validating object identifiers when objects are created, and providing synchronization and consistency for the cached copies of the objects. The object directory can assist a DSVM system in these tasks by storing the information the DSVM system would need to manage each object.

The DSVM system uses an object repository called the *Global Directory of Objects* (GDO) that provides distribution, efficient data retrieval, and scalability. The GDO is logically a table with an entry for each persistently stored object. To ensure that the virtual address range used by a newly created object would not collide with an already existing object, the process creating the object could invoke a service at the node containing the object directory to validate the object's addresses. The service can use an OID which serves as the key field for lookups in the directory. Much additional information needs to be stored in a directory entry according to different distributed system designs. Basically, a global directory is a search structure whose entries describe individual objects. It may contain a large number of entries and the number of entries may grow from relatively few to many. A hash index structure is used in our prototype implementation. In our system, the URL addresses are used as OIDs. If all the addresses of a newly created object do not conflict with the addresses used by existing objects, the service could then register the object by simply creating a new entry in the hash table for the object in the directory. If a conflict occurs, a conflict resolution algorithm can be used so the service can retry to register the object again.

3.4 Treadmarks

Treadmarks [KAZ+96, Tre] is an efficient user-level page-based software DSM system: it provides shared memory between multiple workstations interconnected by a network.

Treadmarks provides a global shared address space across the different machines on a cluster. When shared memory is accessed on one processor, Treadmarks determines whether the data is present at that processor, and if necessary, transmits the data to that processor without programmer intervention. When shared memory is modified on one processor, Treadmarks ensures that other processors will be notified of the change so that they will not use obsolete data values. This takes place between two processors when they are synchronized which greatly reduces the overhead of interprocessor communications. Treadmarks API provides facilities for process creation, destruction, synchronization, and shared memory allocation.

Treadmarks functions and global variables are contained in the header file "Tmk.h" which should be included in any file that calls the Treadmarks software library. Some Treadmarks functions used in our prototype implementation are summarized in Table 3.1.

To avoid data races, Treadmarks provides two synchronization primitives: *barrier* and *locks*. Barriers are global: the calling process is stalled until all processes arrive at the same barrier. Locks are used to manage access to shared resources. A lock enforces one-process-at-a-time access so no process can acquire a lock while another is holding it and only the processor that acquires a lock can release it. Locks are used to synchronize multiple writes to a data item by different processes.

*Tmk_startup(int argc, char **argv)*	Initializes Treadmarks and start the remote processes.
Tmk_exit(int)	Terminates the calling process.
Tmk_malloc(unsigned)	Allocates shared memory between the Treadmarks processes.
Tmk_free(char)*	Releases dynamically allocated shared memory.
Tmk_distribute	Distributes values in private memory on calling process to every other process.
Tmk_barrier(unsigned b)	Blocks all processes until all of them have invoked Tmk_barrier with the same barrier *b* as argument.
Tmk_lock_acquire(id)	Blocks the current pcocess until it acquires the specified lock.
Tmk_lock_release(id)	Releases the lock specified by *id.*
Tmk_proc_id	Variable that contains the current Treadmarks process identifier.

Table 3.1: Treadmarks Application Interface

Treadmarks uses a form of the release-consistency relaxed memory model called lazy release consistency (LRC) which delays the change notification until the next attempt to acquire the same lock. This avoids the overhead of notifying all processors on the lock release. Moreover, Treadmarks uses an invalidate protocol so that this change notification is merely used to mark pages as no longer valid. Actual data updates are only distributed on demand, that is, when a processor attempts to access the stale page.

Treadmarks also has a mode for the multiple-write case where multiple processors are writing to a given page. Multiple-writer mode significantly decrease the performance cost of false sharing, a problem that can occur when two processors are both writing to data items on the same page. Note that if they are actually writing to the same datum the program has a memory race. Treadmarks adaptively selects multiple-writer mode or single-writer mode for individual pages based on the program's behavior [ACDZ97].

Treadmarks is used to provide shared memory between workstations to develop our prototype. The shared memory manager for storing and retrieving documents is distributed across all sites. Treadmarks only provides and maintains shared memory between workstations that are specified in the configuration file ".Tmkrc" during system startup. It does not support the addition of workstations in the middle of the program's execution so some forms dynamic behavior is not possible.

By encapsulating these techniques and properties, a distributed collaborative caching system for efficient document sharing over the WWW is proposed and implemented in this thesis. The system provides a large distributed virtual memory that is shared across a set of browsers in a network without any central proxy

servers. The shared memory is provided and managed by Treadmarks.

Chapter 4

Distributed Collaborative Caching for WWW

Distributed Collaborative Caching (DCC) for WWW is a browser-to-browser document sharing scheme. The collaborative cache environment where browser caches are willing to cooperate is provided by *Distributed Shared Memory* (DSM). DSM provides a large *Shared Virtual Address Space* (SVAS) coordinated by all participating browsers. Retrieved documents are stored in the SVAS of DSM and are visible to all participating browsers. If a cached document is requested a short time later by a participating browser, it will likely be returned from the shared memory. This can reduce document retrieval time, save network bandwidth, and reduce server load by satisfying some requests directly from the shared memory rather than always connecting to the remote server.

The key to building such a distributed collaborative caching system is a directory structure that allows an individual browser to locate documents stored at other

browsers. A *Global Directory of Object* (GDO) is used in DSM system for efficient document retrieval and insertion. The GDO is a hash index structure distributed across all participating browsers and may be replicated.

4.1 Architecture Overview

The goal of our system design is to make multiple browsers cooperate to create one large distributed cache. The documents stored in this cache should be shared and accessed by all collaborating browsers. To achieve this goal, a transparent DSM layer is used between browsers and the Internet. The general system architecture for Distributed Collaborative Caching for WWW is shown in Figure 4.1.

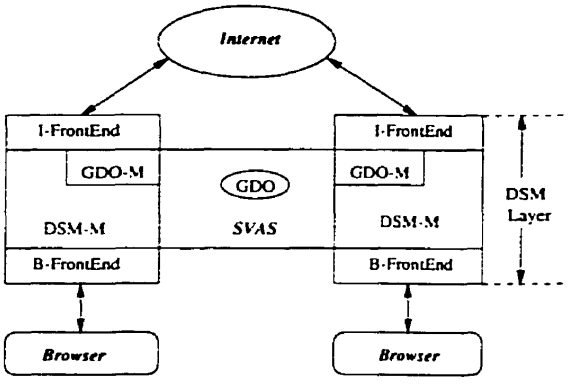

Figure 4.1: Architecture for Distributed Collaborative Caching for WWW

In our system, the DSM layer is composed of several components that communicate with each other to provide document services to browsers. It is responsible for intercepting document requests sent from browsers, relaying requests and documents between the Internet and browsers, and storing retrieved documents. The DSM layer offers a large distributed cache for storing documents and interfaces to

browsers and the Internet. It consists of the following components: (1) The *Browser FrontEnd* (B-FrontEnd) is responsible for the communication between browsers and DSM by using the proxy port setup in the browser. (2) The *DSM Manager* (DSM-M) provides a single virtual memory abstraction among processors at all nodes and maintains shared memory consistency. The shared memory is provided by Treadmarks. (3) The *GDO Manager* (GDO-M) manages a hash index structure (GDO) in shared memory for efficient document insertion and searching. (4) The *Internet FrontEnd* (I-FrontEnd) is responsible for the communication between DSM and the Internet.

The GDO in DSM should provide an efficient and expandable structure for managing documents in shared memory. It can be a linear array, link lists, B+ tree and hash table *etc.* Since hashing is a quick way of predicting the document address in shared memory and can scale easily, a hash table approach is chosen as our GDO structure.

The general functionality of the system components in the DSM layer are characterized as follows:

- **Browsers**

  1. Specifies a proxy server on a certain port.
  2. Interacts with the user.
  3. Waits for the URL requests from the user.
  4. Sends requests to the the proxy server port by using HTTP protocol.

     Note: The proxy server port command is intercepted by the B-FrontEnd.
  5. Displays results for users.

- **B-FrontEnd**

  1. Interacts with the browser through a socket connection.
  2. Listens for the document requests from the proxy port setup in the browser.
  3. Writes the URL requests to the DSM-M through a socket connection.
  4. Waits for the response from the DSM-M.
  5. Sends documents back in HTML format to the browser for display through a socket connection.

- **DSM-M**

  1. Interacts with the B-FrontEnd through a socket connection.
  2. Waits for the URL requests from the B-FrontEnd.
  3. Interacts with GDO-M.
  4. Sends the documents back to the B-FrontEnd.
  5. Interacts with the I-FrontEnd through a socket connection.
  6. Sends the document requests to the I-FrontEnd if the document is not found in the DSM or outdated.

- **GDO-M**

  1. Interacts with the DSM-M.
  2. Waits for the requests from the DSM-M.
  3. Checks and updates the GDO in the DSM for the requested documents.

4. Sends the response (the requested document information or Not Found message) to the DSM-M.

- **I-FrontEnd**

1. Interacts with the DSM-M through a socket connection.
2. Listens for the URL requests from the DSM-M.
3. Interacts with the Internet through a socket connection.
4. Sends the URL requests to the Internet by HTTP protocol.
5. Reads the responses from the Internet.
6. Writes the responses back to the DSM-M through a socket connection.

From the browser's perspective, the system is easy to setup. The only modification to the current WWW environment is to add a transparent DSM layer between browsers and the Internet. The SVAS is distributed among all the browsers connected to the DSM. Correspondingly, the browsers need to be modified so that their document requests are sent to the DSM instead of to the Internet. The proxy server option of the browsers should be filled in the fully qualified machine name (eg., sparrow.cs.umanitoba.ca), and the proxy port number (eg., 8000). After successfully setting up the browser, it can be used normally.

Note that in our system there are no additional proxy servers or other index servers. The DSM layer components exist on each machine with a browser. All browser caches cooperate to provide a large sharing environment for all participating browsers. The DSM-M and GDO-M are distributed across all participating browsers. The B-FrontEnd and I-FrontEnd components function as interfaces between the browsers, the DSM, and the Internet. Without proxy servers in our

system, browser-to-browser collaboration is completely provided by DSM, GDO and two front-end applications. When a browser requests a document, it sends its request to the DSM-M through B-FrontEnd and the GDO-M checks the GDO in shared memory for the requested document. If the requested document exists, it is returned to the browser from the shared memory. If a requested document does not exist in the shared memory, the DSM-M retrieves the document from the WWW server through I-FrontEnd and the GDO is updated with the document details.

Our approach distributes the management of documents across the participating browsers and hence distributes the workload. Access to an origin server for the same document by each browser is no longer required. Since there is no central proxy servers and other index servers in the system, our approach also alleviates the bottleneck and single point of failure problems.

4.2 System Operation

The thesis' goals are to design, implement and evaluate a system that can provide document sharing between browsers in an effort to improve response time for browsers, reduce network traffic and WWW server load, and improve document availability. This section provides a minimally equipped instance to show how our system meets these demands. The system's operational stages are depicted in Figure 4.2 showing how collaboration is achieved between browsers through the DSM. Note also that one of its benefits occurs when a remote server or network link fails.

For simplicity, but without loss of generality, two browsers are used to illustrate

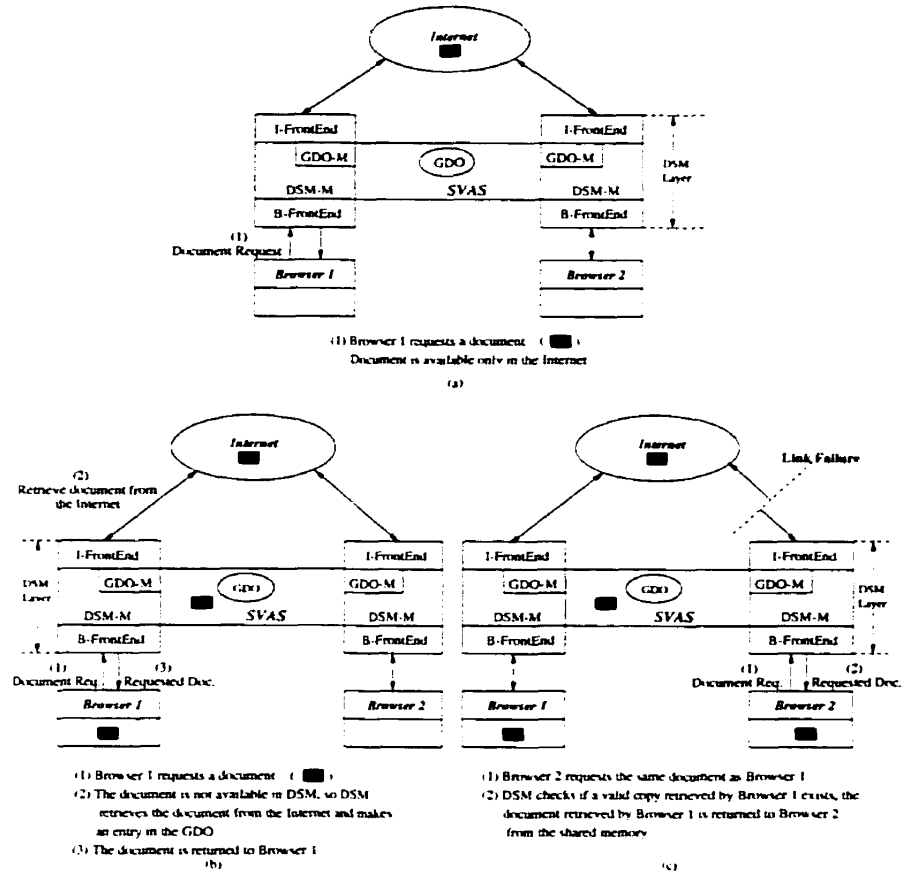

Figure 4.2: Operational Stages

the system operational stages. Before system startup, the names of workstations on which the two browsers are running should be specified in a configuration file called *.Tmkrc* in a "home" directory. Each line of this file must contain the network name of exactly one host. The browsers should be modified to send their requests to a specified proxy port. During system startup, these two workstations are allocated the same shared address space by Treadmarks. At this time, as shown in Figure 4.2a, documents are available only in the Internet and no documents are resident in the DSM. Browser 1 sends a request for a document. The request is sent

to the proxy port setup in the Browser 1 using HTTP. Meanwhile, the B-FrontEnd keeps listening to that port for the requests. When the B-FrontEnd receives a URL request, it sends it to the DSM-M first and no connection to the Internet is made at this time. The DSM-M manages the shared memory across all sites using Treadmarks. When the DSM-M receives the document request, it interacts with the GDO-M. GDO-M checks the GDO in the DSM to determine if there is an entry corresponding to the requested URL. A hash table is used by the GDO for quick and efficient retrieval of documents. Initially, since the document has not been retrieved by any other browser, no entry in the GDO will be found. A "Not found" message is sent to the I-FrontEnd by the DSM-M. The I-FrontEnd sends the URL request to the Internet and waits for the response. After the document is retrieved from the Internet successfully it is sent to I-FrontEnd which is responsible for relaying the document to browser for display. At the same time the GDO-M updates the GDO in the DSM with the document details. A document copy may also be stored in browsers cache. Figure 4.2b shows the request which has been processed by the DSM.

Figure 4.2c illustrates the advantage of having the DSM provide collaboration among distributed browsers. Suppose another browser (eg., Browser 2) sends a request for the same document to the DSM-M through the B-FrontEnd interface. The DSM-M interacts with the GDO-M first to check the GDO for an entry that corresponds to the requested URL. Since the document had been retrieved by Browser 1 and stored in the DSM, an entry is found in the GDO. The document is then validated using some standard validation rules (eg., Apache Server validation rule [Apa]). If the document is valid, it is returned to Browser 2 directly from the

DSM. No connection to the Internet is necessary. If the document in the DSM is outdated, the DSM-M will retrieve it from the Internet. In case of the remote server that "owns" the requested document has crashed or is unreachable, the system returns the document in the DSM with a warning that it is a previously cached copy. This isolates end-users from WWW server and network failures and thus increases the document availability.

4.3 Software Architecture

The DSM layer requires several software components to cooperate to achieve system functions. Figure 4.3 shows the system software architecture and the relationships between the components.

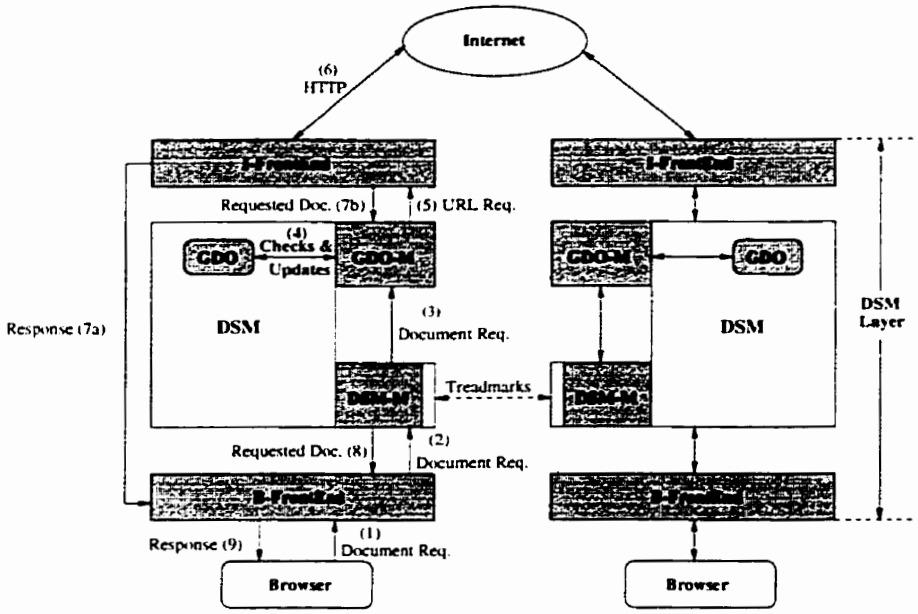

Figure 4.3: Software Architecture

The grayed components are the software modules written to develop the prototype. The other components are existing software and systems that interface with the prototype. The prototype uses the commercially available Treadmarks software package and its Application Programming Interface (API) for providing shared memory between workstations. Some of the API used for this prototype are described in Section 3.4.

The B-FrontEnd functions as a interface between the DSM and the browsers on a particular machine. This software component is coded using Java. It communicates with the browser using the proxy port given in the browser by making a socket connection. After system startup, the B-FrontEnd listens to that port. Once a document request is intercepted by B-FrontEnd, it sends the request to the DSM-M. The DSM-M is a set of Treadmarks API calls that provides a single virtual memory abstraction between workstations. When the DSM-M receives the requests from the B-FrontEnd, it interacts with the GDO-M to check if the requested document is in the DSM already. The GDO-M manages a hash table (GDO) in the DSM. Each GDO entry consists of a URL as an *Object Identifier* (OID) and other related document information. When a document request is received by the GDO-M, it checks the GDO for the document. If the document is found and is valid in the DSM, it is sent to the browser from the DSM directly. If the document is not found or is outdated in the DSM, the request is sent to I-FrontEnd whose task is to retrieve the document from the Internet. The I-FrontEnd functions as a interface between the DSM and the Internet. This software component is also coded in Java. When a request is received by the I-FrontEnd, it sends the request to the Internet using HTTP and waits for the response. The response could be an error message

Software Components	Available (Language)	Written (Language)
Browsers	X	
DSM-M		X (C)
Treadmarks Package	X (C)	
B-FrontEnd		X (Java)
I-FrontEnd		X (Java)
GDO-M		X (C)
GDO		X (C)

Table 4.1: Software Components

or the requested document. The I-FrontEnd sends the response to the B-FrontEnd and the GDO-M if the response is the requested document. The B-FrontEnd then sends the response to the browser for display and the GDO-M updates the GDO entry in the DSM with the document details.

Table 4.1 gives a list of the software components that were available and the components that are written to develop the system.

4.4 System Execution

In the previous sections we have presented our system architecture and system software components. This section gives the detailed execution steps to show how those software components cooperate to achieve the system functions.

Figure 4.4 shows an example of prototype executing in our Lab. A transparent

DSM layer is added between the browsers and the Internet. The DSM is provided by Treadmarks between the workstations. These workstations correspond to the nodes in the DSM and manage the documents in the shared memory. Once Treadmarks starts up, each workstation connected to the DSM will share the address space where documents and the GDO are stored. In our example setup, two browsers (eg., *sparrow* and *swan*) are connected to the DSM. The proxy server option of a browser on each machine is set to port 8000.

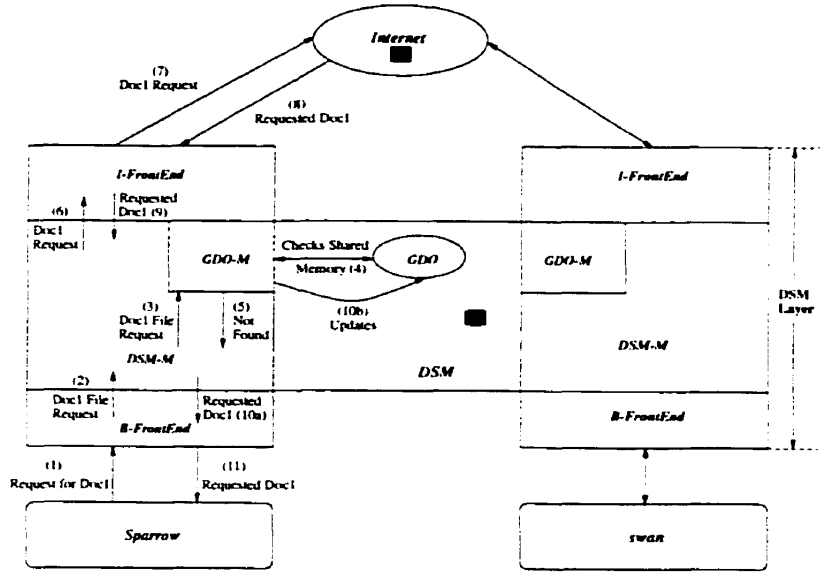

Figure 4.4: Prototype Execution: Stage 1

Since Treadmarks does not support the addition of workstations while the system is running, any participating workstations must be listed in the *.Tmkrc* configuration file before the system starts up. However, additional workstations can be easily added to this file and when the system invokes the DSM the next time they will participate in the shared cache. Browsers can be invoked at any workstation in the network. Because the browsers have been modified to a proxy port, when a

user enters a URL request, it is sent to the DSM instead of to the Internet.

We now describe each of the steps shown in Figure 4.4.

Step 1 A user on *sparrow* requests a document (*Doc1*) by entering the URL or by clicking on a hyperlink of the document. The browser sends the request to the proxy port 8000.

Step 2 The B-FrontEnd listens to port 8000. When the B-FrontEnd receives the request, it sends the URL request to the DSM-M and keeps listening to that port for subsequent URL requests.

Step 3 The DSM-M manages the shared memory. When the DSM-M receives a URL request it then interacts with the GDO-M for the requested document in the DSM.

Step 4 The GDO-M checks the shared memory for the document by searching through the GDO (hash table) with the URL as the key.

Step 5 Initially, since the document (*Doc1*) has not been retrieved by any other browser, the GDO-M cannot find the document in the shared memory. Thus, it replies with a "Not found" message to the DSM-M.

Step 6 When the DSM-M receives the "Not found" message, it sends the document request to the I-FrontEnd.

Step 7 When the I-FrontEnd receives the request, it sends the URL request by HTTP and retrieves it from the Internet. The Internet responds either with the document (*Doc1*) or with an error code.

Step 8 The response is sent from the Internet server to the I-FrontEnd using HTTP.

Step 9 The I-FrontEnd sends the retrieved document to the B-FrontEnd and the DSM-M for storing in the DSM.

Step 10 When the DSM-M receives the document, the DSM-M interacts with the GDO-M. The GDO-M checks the shared memory for the URL. If the document entry is found (this means that the document is outdated in the memory, otherwise, the document will be returned from the shared memory directly in Step 4), the GDO-M updates the entry with the latest content. If the document entry is not found, then the document is stored in the shared memory (GDO) as a new document.

Step 11 When the B-FrontEnd receives the document it sends the retrieved HTML page to the browser using HTTP.

One advantage of making browsers collaborative through shared memory is shown in Figure 4.5. When the communication link with the Internet is down, the browser can still get a cached copy from the DSM, even if it is stale. Because in such a case, no document can be retrieved from the Internet until the link is recovered.

As illustrated in Figure 4.5, during this period another browser (eg., *swan*) with a broken link with the Internet requests for the same document (*Doc1*). The request is sent to the DSM-M through the B-FrontEnd. The DSM-M interacts with the GDO-M to check the shared memory for the requested document by searching the GDO. Since the document had been retrieved by *sparrow* and is stored in the shared memory, it is found in the DSM. If the document is valid using some standard validation rules, it is returned to *swan* from the shared memory directly without connection to the Internet. If the document is outdated in the shared memory and

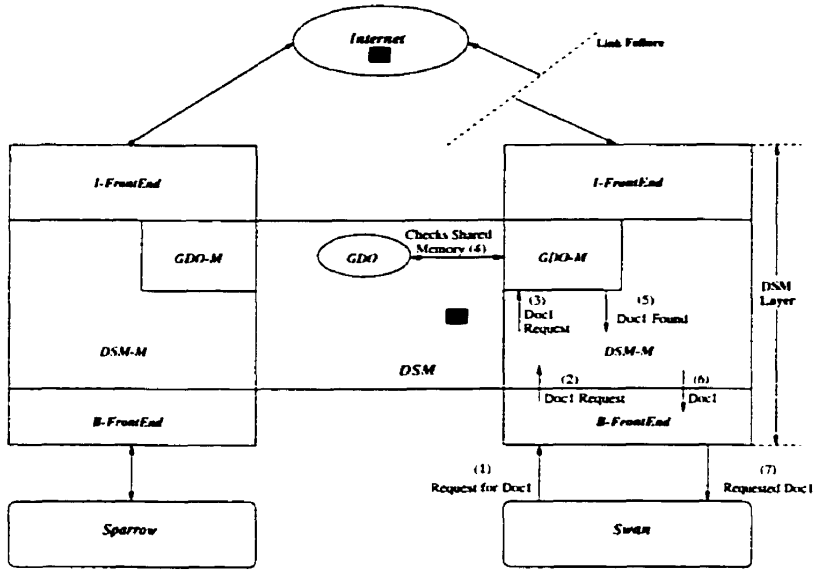

Figure 4.5: Prototype Execution: Stage 2

the Internet is unreachable, the document is sent to the browser with a warning that it is a cached copy. A stale document is often better than no document at all.

The scalability of the system is good because no central mapping or proxy servers are applied. Since the SVAS is provided by the collection of all participating workstations, the more workstations that participate, the larger SVAS grows. The SVAS is distributed among these participating workstations and each workstation manages its own address space. Since no central proxy server is needed in the system, there is no bottle neck problem caused by a central structure. Any number of workstations can be involved thereby providing additional support to the shared memory cache. This is accomplished by simply adding each workstation's network name in *.Tmkrc* file. When the system starts up, Treadmarks allocates a global shared address space across all participating workstations. Any data item at a virtual address appears at the same address and is visible to all workstations. When

a data item in SVAS is accessed by a workstation, there is no need to move this data item to make it accessible by the programmer. Instead, Treadmarks will move it implicitly. Therefore some overhead will be incurred by Treadmarks. This observation is shown in our performance tests (see Chapter 5).

However, since no central mapping or proxy servers are used, some issues arise while linking browsers in several networks (eg., Computer Science Department, Electrical Engineering Department, *etc.*). In this case, browsers in one local network can form an intranet (say, Computer Science Department) and another group of browsers in another network (say, Electrical Engineering Department) can form another intranet. Each local network manages the invocation of browsers within its own domain. The GDO is fragmented and distributed across different networks. The collaboration can be achieved by linking all intranets. However, this results in some network load on the Internet.

4.5 System Implementation

4.5.1 Implementation Environment and Languages

Our system can execute on a network of workstations. It can run over Ethernet, FDDI, and ATM networks on different UNIX platforms, such as DEC Unix, HP-UX, SunOS, Solaris, AIX, Ultrix, IRIX and Linux. The system in our laboratory is setup on Intel x86-based PCs under Linux. It can include any number of participating workstations.

Both C and Java are chosen as the programming languages. The B-FrontEnd and I-FrontEnd modules are implemented using Java, and the GDO, GDO-M and DSM-M modules are implemented in C. These components constitute the DSM layer and communicate with one another to provide sharing services among browsers.

We choose Java for the B-FrontEnd and I-FrontEnd interface implementation language for the following reasons. First, Java has emerged as a network language breaking the barriers of architecture and operating system dependence so it allows for a high degree of code portability. The same program can be executed securely on multiple computers, over any operating system as long as the Java Virtual Machine runs on the underlying system. Second, Java has solved problems such as security and automatic generation of documentation while staying faithful to the object-oriented paradigm. Through inheritance and polymorphism, one can develop reusable code making good use of previous programming efforts and laying the ground work for the development of future applications.

However, a completely portable Java program has some drawbacks: speed and the inability to access platform-specific services. Since the shared memory is provided by Treadmarks, which is implemented as C library functions, we choose C as the GDO, GDO-M and DSM-M implementation language. This increases the system performance to some degree, which is one of purposes of this thesis. Therefore, the use of Java for system front end implementation and C for main system operation meets our demands best.

4.5.2 Hash Function for GDO in DSM

Quick information retrieval is obviously crucial to our system performance. There are several different data structures for organizing the information in the DSM, such as linear array, link list, B+ tree, or hash table. The performance differ greatly for different scenarios. Since we need to potentially store large amounts of information in the DSM and retrieve this information quickly, a hash table is chosen as our GDO structure.

A hash table takes a key value and applies a function to produce an address. The resulting address is used as the basis for searching for and storing data items. Two different keys may be transformed to the same address which causes a *collision*. There are several ways to reduce the number of collisions. One approach is to hold more than one data item at a single address. Addresses that can hold several data items are called *buckets*.

There are several good reasons for using hashing in our system. First, hashing, unlike indexing (eg., B+ tree), requires no extra storage for the index. Second, hashing is the most effective accessing method. Hashing provides a faster insertion, deletion, and retrieval of records than other accessing methods such as through index (B+ tree) or sequentially [FZ87]. Hashing usually takes less than two accesses to find a record. Third, hashing is extensible because the entries in the GDO are added dynamically based on user demands so the data structure for the GDO should be able to accommodate the continued insertion of documents. The hash table can be an index with pointers to buckets that contain the actual data records. If more than one keys is transformed to the same bucket, an additional record is created and chained to the existing record through a pointer. Finally, the

randomizing characteristic of hashing fits the WWW scenario well. With hashing, the addresses generated are random. There is no obvious connection between the key and the location of the corresponding record. The perfect result for those generated addresses is that there are no collisions. However, this is hard to achieve. In our system, the documents are stored in the shared virtual address space randomly. URLs are transformed to the keys for locating documents in the shared memory. Within WWW, the URLs are unique and appear to be random, therefore, using URLs as keys can avoid collisions to some degree. This characteristic of WWW browsing makes hashing a good choice [Squ].

The following data structures defines the hash table (GDO) in the DSM.

```
typedef struct hash_entry
{
    char *key;                //pointer to hash key (URL)
    struct hash_entry *next; //pointer to next hash entry for this
                              //hash value
    char *document;           //pointer to hashed document
    int document_size;        //hashed document size
    ...                       //other information of hashed document
}HASH_ENTRY;

typedef struct hash_table
{
    int no_buckets;        //number of buckets in the HASH TABLE
    HASH_ENTRY **buckets; //pointer to HASH TABLE (array of pointers)
}CONTROL_INFO;
```

The HASH_ENTRY structure contains the information about the retrieved documents in the DSM. It contains pointers to the hash key and hashed documents. If there is more than one URL that hashes to the same bucket, an additional HASH_ENTRY is created and chained to the existing HASH_ENTRY through the

next pointer in the structure. Some other related document information is also defined in this structure. This information includes header definitions of a HTTP message header, such as document size, document type, document expiry date, last modified date, retrieved date, *etc.* This information is used to update and check the document status in the DSM.

The CONTROL_INFO structure contains the pointer to the hash table itself. It contains a number of buckets in the hash table and a pointer to an array of buckets. The pointers in the array of bucket point to the HASH_ENTRYs.

A schematic representation for the two structures is shown in Figure 4.6. When a document is to be stored in the hash table, its corresponding URL is hashed to fit within the number of bucket addresses created. Hashing transforms a URL into an integer that serves to locate an element of the bucket array. The bucket contains a pointer to a HASH_ENTRY and the HASH_ENTRY contains pointers to the hash key, the hashed document, and other information of the hashed document. If there is a collision in the same bucket, an additional HASH_ENTRY is created and chained to the existing HASH_ENTRY.

Retrieved documents are stored in the DSM with URLs serving as keys for searching and inserting into the hash table (GDO). When a browser retrieves a document, the URL is sent to the DSM and used as a key for mapping the document in the hash table. When the DSM-M gets the URL request, a hash function is performed to transform the URL into an address. The resulting address is used as the location for searching and storing documents within the hash table in the DSM. If a key is found with that address, it means that the document has already been stored in the DSM. The document will be sent to the browser if it satisfies

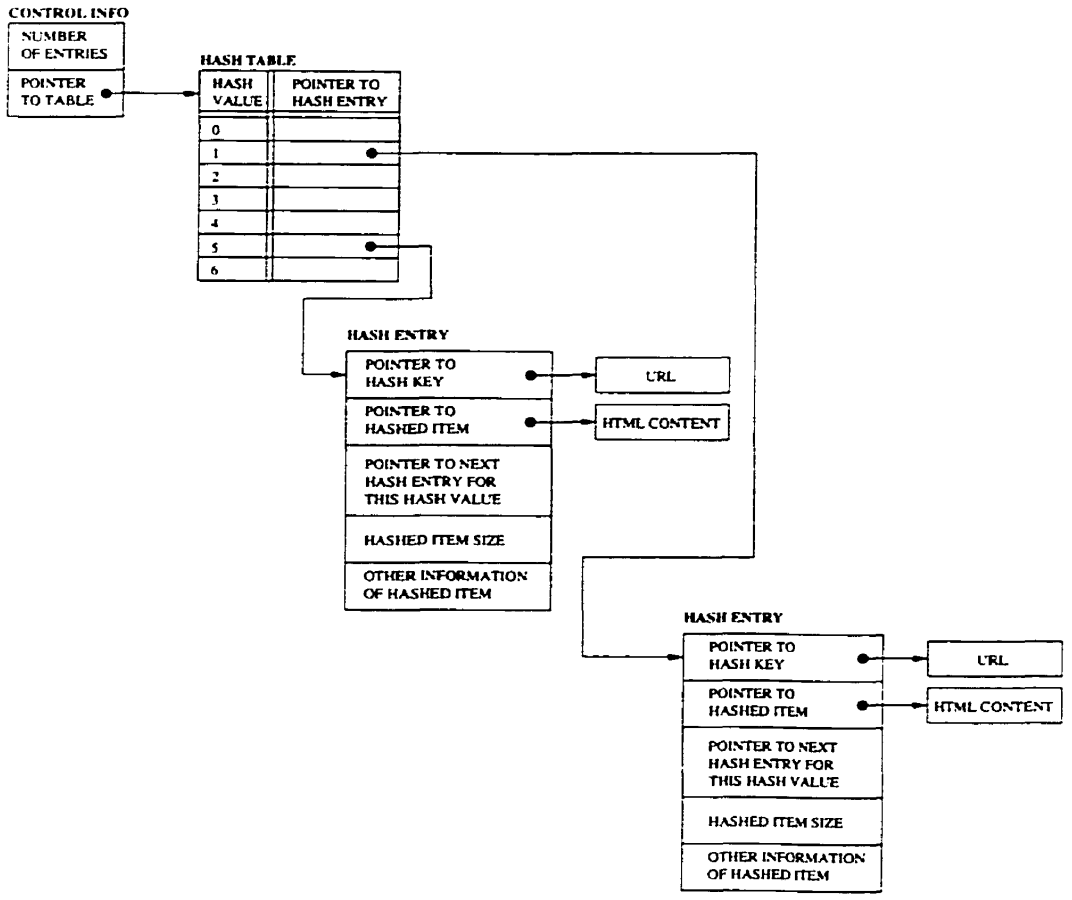

Figure 4.6: Schematic of Hash Table in DSM

the validation rules. Otherwise, the URL request is sent to the Internet. When the GDO-M gets the document from the Internet, it inserts the document into the hash table with the associated URL as its key.

The goal for any hash function algorithm is to spread out records as uniformly as possible over the range of addresses available. However, a perfect hashing algorithm without collision is hard to achieve. The following hash function used in this thesis returns a well distributed address within the address range. The similar function is also used in Squid [Squ] and has proven to be a good hash function with small

number of collisions.

```
#define HASH_SIZE 7951                    // Prime number < 8192
int hash_url(char *url)
{
    unsigned i, j, n, addresss;
    j = strlen(url);
    for(i = j/2, n = 0; i < j; i++)       // Sum up the last half of
        n ^= 271 * (unsigned) url[i];     // the URL string
    i = n ^ (j * 271);                    // Do more randomizing
    address = i % HASH_SIZE;              // Divided by the size of the
                                          // address space
    return address;                       // return the hash address
}
```

The first step of this function is to sum up the last half of the URL string. This is carried out by performing *bitwise exclusive or* operation on numeric form of each character in the last half of the URL string. The numeric value of each character is enlarged by multiplying a number (say, 271) to increase the randomizing effect. After the loop is finished, a "random" number is created that can be used to create the physical hash address in the following steps. The second step is to perform another *bitwise exclusive or* operation on the "random" number obtained in the first step with a numeric value which is the length of the URL string multiplied by some number. The purpose of this step is the same as explained in the first step in that it does more randomizing. The final step of the function is to divide the number obtained in the second step by the size of the address space (HASH_SIZE) and then take the remainder. The remainder will be the home address (HASH VALUE in Figure 4.6) of the record.

The HASH_SIZE which determines the size of the address space does not have to be any specific size as long as it is big enough to hold all of the records. It is

used as divisor and usually a prime number because primes distributes remainders much more uniformly than nonprimes [FZ87]. The remainder produced is a number between 0 and HASH_SIZE - 1 and will be the address of a record.

The hashing method used in our system is a static hash algorithm because it uses a constant sized table. Some dynamic hash algorithms like Linear Hashing [Lit80, LNS93] or Extendible Hashing [Sch81] may provide a better performance for our system but it is left as an area for future research.

4.6 DCC Compared to Earlier Prototypes

DCC has some advantages compared to earlier cache prototypes described in Chapter 2.

DCC provides a distributed approach to manage the documents in a distributed shared memory. It is based on a directory structure called Global Directory of Objects (GDO) to provide services to the collaborative browsers. GDO is distributed across browser machines and may be replicated on several browser machines in the DSM. The basic idea behind GDO is similar to RPD approach in CRISP but without central mapping servers, therefore, GDO provides a better reliability and scalability.

Compared to Squid, DCC fully distributes the workload across all participating browsers by avoiding any hierarchy since a hierarchical structure still exhibits a centralized "nature" that will limit scalability and have a single point of failure problem.

The idea of DCC comes from WWW-DSVM but it takes the concept a step

further by eliminating proxy servers in the system. DCC makes each browser machine a node in the DSM and all browser caches cooperate to provide a large sharing environment for all participating browsers. The DSM system is provided and maintained by Treadmarks across all sites. GDO is used by the DSM system to provide distribution, efficient data retrieval, and scalability for the system. It is distributed across all participating browsers. Instead of using proxy servers as DNS and GDO interface for participating browsers in WWW-DSVM, there are two front-end applications are used by each browser in the DSM in our system. The *B-FrontEnd* (Browser FrontEnd) is used for the communication between browsers and the DSM, and the *I-FrontEnd* (Internet FrontEnd) is used for the communication between the DSM and the Internet. Browser-to-browser collaboration is provided by DSM, GDO and two front-end applications without the requirement for proxy servers. When a browser requests a document, it sends its request to DSM through B-FrontEnd and the DSM manager checks the GDO in shared memory for the requested document. If the requested document exists, it is returned to the browser for display from the shared memory. If the requested document does not exist in the shared memory, the DSM manager requests the document from the WWW server through I-FrontEnd and the GDO is updated with the document details.

DCC distributes the management of documents across the participants and hence distributes the workload. Access to a server for the same document by each browser is no longer required. Since our approach is fully distributed and without any central servers it has good reliability and scalability when compared to other approaches discussed above.

Chapter 5

System Performance

One goal of this thesis is to evaluate the performance our design for collaboration among multiple browsers. Many researchers [ASA+95, TDVK98, FCA98, GRC97, GCR97, DMF97, AFAW97] have shown that WWW caching can save network bandwidth and reduce latency thereby improving the performance of the whole WWW. There are several ways of evaluating the performance of the WWW caching. Some of the approaches use information concerning the utilization of computational resource, such as memory, disk space, CPU usage, *etc.* Other approaches consider bandwidth utilization, cache hit ratios, or latency perceived by end users. However, most published researches on WWW caching are concerned only with improving the WWW cache hit ratio. Improving only the hit ratio and ignoring the actual retrieval latency experienced by WWW users may not be a good idea because the web users care little about hit ratios. On the other hand, high hit ratio does not necessarily mean fast response time. For example, there is no benefit in caching local documents which may, in some cases, be retrieved faster from the origin server

than from the cache despite the higher hit ratios.

Our system is designed to improve the performance for WWW document retrieval and provide high scalability and reliability. The previous sections have already shown that our system has a good scalability and reliability. The objective of this section is testing performance of our system architecture configurations in terms of response time, that is, how long it takes to serve end user requests. First, some performance results from related studies are summarized, then our measurements of the response time is presented and analyzed under different conditions. These performance results are essential for designing, modeling, and tuning the performance of WWW caching.

5.1 Background

The interpretation of some commonly used WWW caching terms are given below.

- **Hit** A client request that is successfully satisfied without transmitting the document content from the WWW server.
- **Miss** A client request that is not a hit.
- **Hit Ratio (Hit Rate)** The ratio of the number of HIT requests to the total number of client requests.

Hit ratio is an important measurement because a higher hit ratio means that the cache is performing better. However, the results show that WWW proxy caching has a 30-50% maximum hit ratio no matter how it is designed. Gadde

et al. [GCR97] evaluate the CRISP cache using a simulation of twenty-five days of the DEC traces [DEC96]. They see a sharing hit ratio of 45% for an 8GB cache. They also compare the performance of Harvest [CDN+96] with CRISP. They replay a one day DEC trace through three configurations: Shallow Harvest, Flat Harvest and CRISP/CSD. Shallow Harvest consists of four child caches with a common parent. Each server has a 16MB memory cache and a 100MB disk cache. Flat Harvest is identical to Shallow Harvest but without the common parent. CRISP/CSD is identical to Shallow Harvest, except that the parent is replaced with a mapping server. They show that both Harvest configuration and the CRISP configuration achieve a hit ratio of 38-40%. There is very little performance difference in these configurations.

Duska *et al.* [DMF97] present an analysis of access traces collected from seven proxy servers. Their results show that 2GB to 10 GB second level cache yields hit ratios between 24% and 45% with 85% of these hits due to sharing among different clients. The hit ratio of the third-level cache is 19% because lower-level caches filter locality and lower-level sharing from its requests stream. The authors argue that caches with more clients exhibit more sharing and thus experience higher hit ratios.

Abram *et al.* [ASA+95] perform a simulation study on three workloads. The results show that a proxy has an upper bound of 30-50% hit ratio given a theoretical infinite size cache. They also find that the caching proxy hit ratio declines as the Web browsers caches fill over time. The reason is that when Web browsers use their own caches, the proxy acts as a second level cache.

Higher hit ratios are achievable for large communities and large caches. Tewari *et al.* [TDVK98] configure the system as a three-level hierarchy with 256 clients

sharing a L1 proxy, eight L1 proxies (2048 clients) sharing a L2 proxy, and all L2 proxies sharing an L3 proxy. They report the hit ratio from 50% for L1 to 62% for L2 and 78% for L3 in the DEC traces. Their study also supports conclusion in Duska *et al.* [DMF97] that cache architectures that scale are important because increasing the number of users sharing a cache system increases the hit ratio achievable by that system. Another pattern shown in these studies is that Web cache hit ratio grows with the cache size but the gain in hit ratio becomes smaller and smaller until an appreciable improvement is noticed. As reported by Duska *et al.* [DMF97], DEC hit ratios reach 41.1% for a 20GB cache and increase very slowly to 42.1% for a 100GB cache. What is a perfect cache size for proxy is still an open research area. However, for the collaborative cache in our system, the shared cache grows with the participating users.

Unfortunately, these studies did not show the response time experienced by WWW users under different cache designs. However, improving the response times for WWW users is one of the main reasons for using WWW caching. The following sections present our experimental results on response time obtained using our system.

5.2 Testing Methodology

It is difficult to model a typical response time because it depends on many factors; such as speed of the Internet connection, geographic location, the link speed between the cache and its users and so on. However, a response time measured on the same network is an important metric for analyzing the benefits of the WWW

caching.

The following terminology is used to refer to the three types of latency we measured in our lab. *Direct latency* is the time it takes to receive a document and all its associated images directly from the WWW servers specified in the URL's for those objects. *Proxy latency* is the time it takes to retrieve a document and all its associated images from the DSM when none of those objects have been previously cached in the DSM. *DSM latency* is the time it takes to receive a document and all its associated images from the DSM that already has those objects cached.

For our tests, the main performance data collected are latency. We performed three experiments and only HTTP requests are considered. The purpose and design of our three experiments are described below:

Experiment 1:

The purpose of Experiment 1 is to measure the three types of latency in a real environment. After starting up our system, it just "sits" there at the background and intercepts the requests sent out from browsers. For this purpose, we obtained a random set of URLs by invoking the "random link" feature of Yahoo [Yah]. We eliminated any URLs that did not use the *http* scheme. First, we ran our system to fetch these documents, extract the URLs for their associated images, and determine the size of each document and image object. For ease of analysis, we then threw out any documents that were referenced more than once and those unreachable documents, which left us with 70 documents on which to perform this experiment.

Experiment 2:

The purpose of Experiment 2 is to measure the three types of latency using trace-driven simulations. Traces provide more realistic real WWW client access patterns

	Total	272	B19
http	568181	65764	502417
gopher	5236	727	4509
ftp	2358	514	1844
Queries	7509	1129	6380
Others	7888	110	7778

Table 5.1: Distribution by Protocol

from which to get more meaningful averages of latency values. The traces used in our experiments were collected by Computer Science Department of Boston University and are publicly available [BUWT]. The trace data was collected from 21 November 1994 to 8 May 1995 in two different environments - Room 272 and Room B19. Table 5.1 summarizes the distribution of requests among different access protocols [CBC95]. The row Queries refers to requests containing 'cgi-bin' in the URL.

For our experiment, we only used the data collected in Room 272 during February 1995. For ease of analysis as in Experiment 1, we eliminated any requests that do not use *http* protocol. We then threw out any documents that were referenced more than once and those unreachable documents to ensure the validity of the trace file. leaving us with 482 documents on which to perform this experiment and next experiment. Then we performed this experiment on only one machine.

Experiment 3

The purpose of Experiment 3 is to measure the effect on latency by adding more

machines. This test is similar to Experiment 2, except that it was performed on three machines instead of one. These three machines read from the trace file at the same time and the three types of latency results were measured for each machine.

For our experiments, the DSM latency and proxy latency are measured by our system. Another program is written to determine the direct latency for each document object and image object. These programs ran on the same hardware and software platforms in our laboratory and on the same subnet. In all our presentations of results, we have not made any effort to eliminate values that might be considered statistical abnormality. This is because seemingly abnormal results are typically due to bursts of uncontrollable external network activity, which is an important aspect of the environment and should be taken into consideration in analyzing the performance of the WWW.

5.3 Experiment Results

5.3.1 Experiment 1

For this experiment we used Netscape Communicator as our browser. The document requests were sent out from browser to our system. Each type of latency was measured five times for each document and the average was taken of those five trials. All trials for a single document and image were done consecutively so that all data for any such document was taken under as similar network conditions as possible. Figure 5.1 shows the mean of each type of latency across all documents.

This figure illustrates that the average response time of a HIT in the DSM is

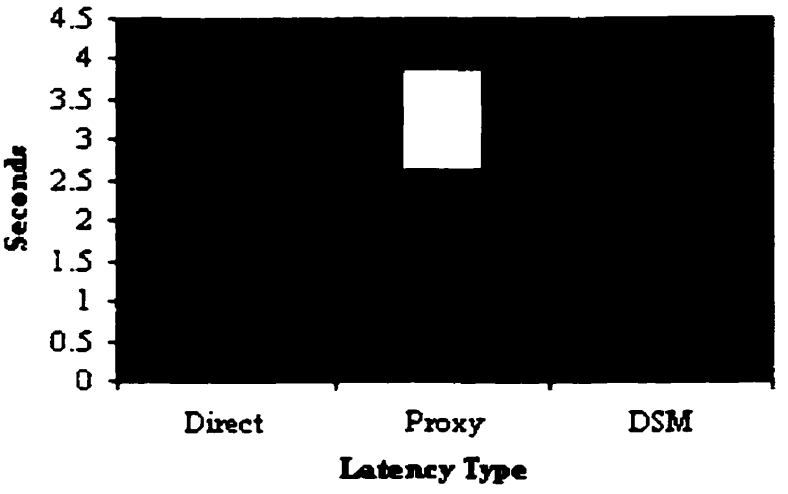

Figure 5.1: Three types of latency across all documents

decreased dramatically compared to direct connection to the Internet. But for a MISS, the average response time is greater than that of direct connection. Therefore, the average improvement by the DSM depends on hit ratio in the DSM.

Another factor that may affect the response time is the document size. Figure 5.2 shows mean response time on different document size.

This chart shows that the response time increases with document size for direct latency and proxy latency but there are some abnormality. This increase is dramatic for large documents, for example, greater than 128K but for DSM latency, there is no evident changes with different document size. Note that those abnormal results follow a similar pattern because every miss in the DSM uses an external network to retrieve documents. This chart also shows an important aspect that the network activity has a significant effect on response time. On the other hand, the DSM latency for a hit in the DSM does not vary much because the external network is not used if we get a hit in the DSM.

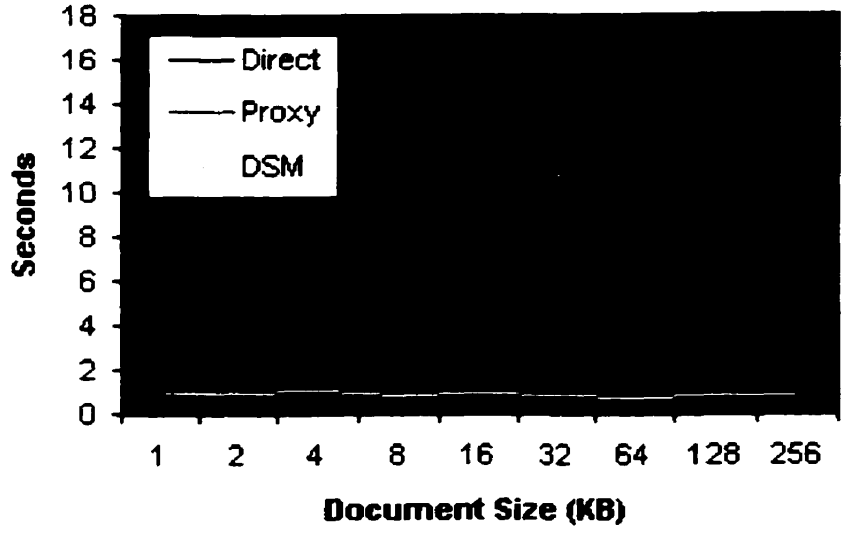

Figure 5.2: Document Size and Latency

5.3.2 Experiment 2

We performed this experiment using trace files. The document requests are contained in trace files. Another program was written for processing the trace files to extract URLs and send the requests to our system. Figure 5.3 shows average latency across all data for different types of latency.

Figure 5.3 is very similar to Figure 5.1. However, there are some slight difference on response time for direct latency and proxy latency. This is because these two types of latency are dominated by factors, such as network traffic, WWW server load, *etc.*. For DSM latency, however, there is relatively little change in response time. Another difference is that all of the three types of latency obtained in this experiment are smaller than those obtained in Experiment 1. The reason is the tests are performed using trace files instead of using browsers as in Experiment 1 so there is no browser overhead.

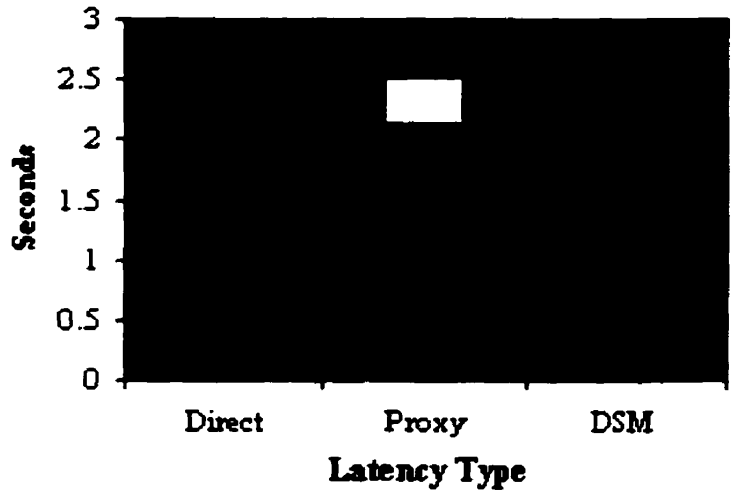

Figure 5.3: Three types of latency across all documents

Figure 5.4 depicts the average latency on document size. Again, this chart is similar to Figure 5.2. Both of these two charts show that document size has little effect on DSM latency, while for direct and proxy latencies, response time increases with document size, especially for large documents.

5.3.3 Experiment 3

Recall that the purpose of Experiment 3 is to measure the effect on latency by adding more machines to the system. We performed this experiment on three machines at the same time in our lab. The experiment results are shown in Figure 5.5.

This chart shows that adding clients numbers may incur some overhead to all three types of latency. This overhead may be more to direct and proxy latencies than to the DSM latency. One reason is that running more clients at the same time will use more network bandwidth, while network activity is one main factor that affects response time as already shown in Experiment 1 and Experiment 2. Since

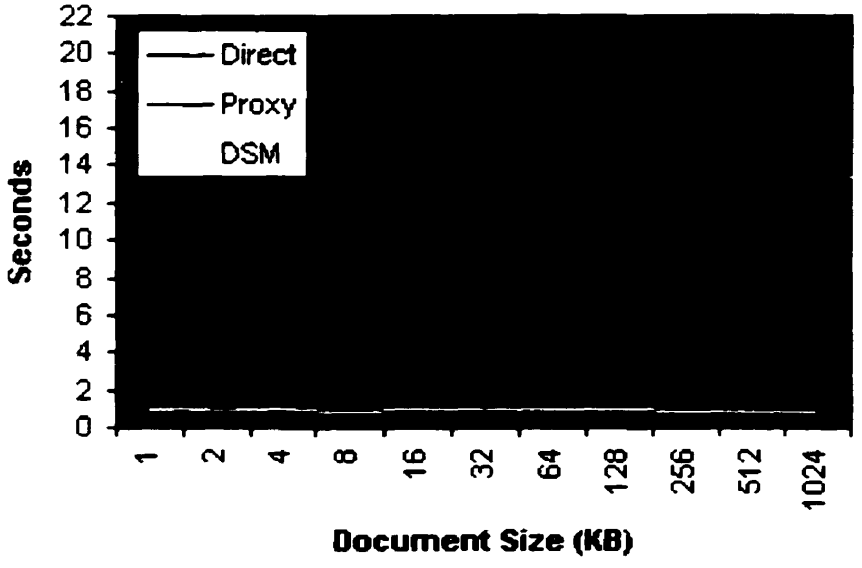

Figure 5.4: Document Size and Latency

hits in the DSM do not use any external network the DSM latency is affected less than the other two types of latency. However, when more clients get the documents from the DSM at the same time will result in some overhead to TreadMarks.

5.4 Discussion

Note first that all our three experiments show similar results. The average direct latency and proxy latency are much higher than that of DSM latency. For large documents this becomes worse. Further, direct latency and proxy latency will be affected by uncontrollable external network activity. Moreover, sometimes they cannot get documents back due to network or server failure. However, this does not apply to the DSM latency because for a hit in the DSM, the document can be sent back to browser from the DSM without using external network and WWW

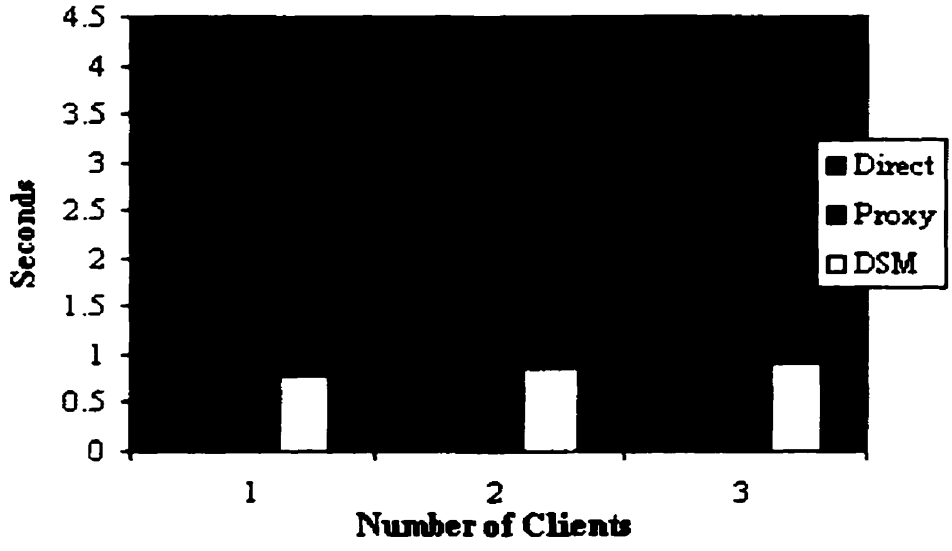

Figure 5.5: Latencies and Number of Clients

servers. Another observation is that although more clients may cause a little more overhead to the system, higher number of clients increase the hit ratio so better response time is expected for all clients.

We should also note that these latency values are only meaningful to average if one considers the workload to be one in which the documents studied in our three experiments are accessed with equal probability. Any more general interpretation of one of our average values is not accurate unless one expects the distribution of latency values for the workload to be similar to that found in a realistic client environment. In the case of proxy and direct latencies, it is likely we have not achieved this goal, as these latencies are very dependent on the distance to the hosts and the time of access, and we have made no attempt to make the distribution of distances to hosts to match that of any sort of typical client. On the other hand, since DSM latency does not involve any communication with the host specified in

the URL's there should be no correlation between host distance and latency.

Although proxy latency is greater than direct latency, we believe that caching is still worthwhile. First, one must consider that the greater cost of a document miss in the DSM is compensated for by document hits in the DSM so the client experiences less latency. For the workloads presented in our experiments we calculate the average latency experience using direct access (this is called the *average direct latency* (ADL)). Our results for the DSM latency are then measured as a percentage of this figure. Thus, for Experiment 1 the average DSM latency is only 35% of ADL while for Experiment 2 it is 37% of ADL. Second, the benefits of caching go beyond reduction of document retrieval latency for cached objects. Caching also ensures reduced use of network bandwidth and reduced server load. These may, in turn, provide savings in document retrieval latency for all objects.

Thus, we can make the following comparison between our DSM caching system and a single caching proxy. Our system permits sharing of multiple caches among many clients through the DSM so we expect a higher cache hit rate. Since documents that hit in the cache take less time to retrieve than ones that miss, this will decrease our relative latency. The increased cache hit ratio will also translate into less use of network bandwidth and less load imposed on remote servers.

Chapter 6

Conclusions and Future Work

In this thesis, a Distributed Collaborative Caching for WWW system is designed and implemented. The system provides a WWW document sharing environment between browsers. The sharing environment is supported by a distributed shared memory across all participating browsers. The purpose of building such a collaborative system for the WWW is to help to reduce network load and server load, which in turn improves the WWW performance as a whole. This goal is achieved in this thesis by servicing some documents in a *Distributed Shared Memory* (DSM). The performance in terms of response times is measured under different conditions. The results show the benefits of the system. The response time for the WWW users is greatly improved by the use of our system. The average response time for hits in the DSM is only 35% of average direct response time. This is significant not only because the response time for WWW end users is greatly improved, but also because it improves the availability of documents for WWW end users. Even if a remote server is unreachable, the user can still get a copy of the document

from the DSM. The advantages of the system comes at a penalty. For misses in the DSM, the average response time is higher than that of direct connection. However, the advantages outweigh the disadvantages because the average response time for all hits and misses is still lower than the average direct response time. On the other hand, the advantages of the system go beyond reduction of response time. It saves network bandwidth and reduces server load, which in turn provides savings in document retrieval latency for all objects over the Internet. By sharing documents between browsers, the system also alleviates problems due to network link failures and WWW server failures.

We have shown that a higher hit ratio in the DSM will provide better performance. Therefore, the system would like to accommodate multiple clients because by increasing the number of clients sharing a distributed memory cache thereby increasing the hit ratio. Thus, the system may be more useful in some application environments, such as in a corporate intranets where a group of people work with similar sets of WWW documents or for an Internet document space [Doc] where WWW users can work together on the same documents. Once a document is retrieved from the Internet, the other browsers then retrieve the document locally from the DSM.

However, there are some limitations of the system at this time:

- The system does not support heterogeneous machine architectures. This is due to the characteristic of the DSM where nodes in DSM system share pages of virtual memory.
- The system does not support dynamic management for shared memory. This means if a workstation wants to join the services of the system, its full Internet

address must be specified in the configuration file ".Tmkrc" during the system startup. This is required by Treadmarks.

- Our workload for analyzing the system performance is small. Actually, it is difficult to characterize WWW traffic (eg., day time or late hours) and access patterns (eg., document size, document types, *etc.*). Different aspects of access patterns may have significant effects on WWW cache performance [AFAW97, DMF97]. Therefore, extensive traces of document retrieval patterns would provide a more realistic workload from which to get more meaningful performance results.

- The number of clients is small. Only three workstations are used to test the system performance in our lab. Although overhead was tested by adding more clients as in Experiment 3, we are uncertain how a much larger client population would affect the results? Some researchers [DMF97] show that a higher number of clients result in higher hit ratios, which in turn improves response times for all clients. Additional experiments are required to determine how much overhead is incurred by the system when adding more clients.

Although these limitations exist, our system is shown to effectively improve the performance for WWW document retrieval. Some possible continuous work on the system are suggested below:

- Provide a partitioned GDO structure with partial replication [MGB96, GCR97]. By partitioning the GDO into fragments, replicating the fragments and distributing the fragments and their replicas across the nodes in the system, the

load placed on the GDO can be distributed. Further, the reliability of the system can be enhanced and the consistency of the directory can be maintained at low cost.

- Use more hash tables for storing different types of documents. At this time, only one hash table is used for storing the different kinds of documents. A better solution is to store the different document types in different tables. For example, storing HTML files in one table while storing image files in another table. This will increase the searching speed and the hash tables have better extensibility.

- Use dynamic hash algorithms like Linear Hashing [Lit80, LNS93] or Extendible Hashing [Sch81]. The hashing method used in our system is a static hash algorithm because it uses a constant sized table. Since we need to potentially store large amounts of information in the DSM, a hash table can grow and shrink from its initially allocated size to accommodate the continued insertion and deletion which will provide better performance for our system.

- Apply pre-fetching technique [DP96] to the system. Pre-fetching periodically refreshes cached documents so as to reduce staleness and response latency when the user requests them. The documents that are related to the requested URL are also pre-fetched in advance before the user explicitly requests them. Pre-fetching can reduce response latency but also wastes some network bandwidth if the pre-fetched documents are not ever requested by the clients.

- The system can be used as the first layer in a hierarchy caching system, such as Harvest [CDN+96] and its successor Squid [Squ] caching system. As in Figure 2.3, each stub network consists a proxy caching server with a set of browsers connecting to it in a local network. Some proxy caching servers in stub networks are connected to a proxy caching server in regional network. However, the centralized nature of proxy caching servers has some limitations as described in Section 2.2. Instead of using a proxy caching server in each stub network, our system can be used to provide collaborations among browsers in a stub network. Since there is no centralized servers anymore, it distributed the workload among browsers and alleviates a single point of failure problem.

In summary, our system is effective and efficient to reduce network traffic and WWW server loads, thus improving the response time for all objects on the WWW. It also alleviates network failure and WWW server failure problems which give users greater benefits within the WWW.

Bibliography

[ACDZ97] C. Amza, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Software Dsm Protocols that Adapt between Single Writer and Multiple Writer. *Proceeding of the Third International Symposium on High-Performance Computer Architecture*, pages 261 - 271, February 1997.

[AFAW97] G. Abdulla, E. Fox, M. Abrams, and S. Williams. WWW Proxy Traffic Characterization with Application to Caching. *Technical Report, TR-97-03, Computer Science Dept., Virginia Tech*, March 1997.

[Apa] Apache Proxy Server. http://www.apache.org.

[ASA+95] M. Abrams, C. Standridge, G. Abdulla, S. Williams, and E. Fox. Caching Proxies: Limitations and Potentials. *In 4th International World-Wide Web Conference*, pages 119 - 133, December 1995.

[Blo70] B. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communications of ACM*, 13(7):422 - 426, July 1970.

[BPG95] K. Barker, R. Peters, and P. Graham. Distributed Shared Virtual Memory for Interoperability of Heterogeneous Information Systems. *OOPSLA Workshop on Interoperable Objects - Experiences and Issues*, October 1995.

[BUWT] Boston University Web Trace. http://ita.ee.lbl.gov/html/contrib/BU-Web-Client.html.

[CBC95] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of Www Client-based Traces. *Technical Report BU-CS-95-010, Boston University*, July 1995.

[CDN+96] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrel. A Hierarchical Internet Object Cache. *In Proceedings of USENIX Technical Conference*, pages 153 - 163, January 1996. http://excalibur.usc.edu/cache-html/cache.html.

[CLF94] J. Chase, H. Levy, and M. Feeley. Sharing and Protection in a Single Address Space Operating System. *ACM Transactions on Computer Systems*, 12(4):271 – 307, November 1994.

[DEC96] Digital Equipment Corporation. Digital's Web Proxy Traces. 1996. ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html.

[DHS93] P. Danzig, R. Hall, and M. Schwartz. A Case for Caching File Objects Inside Internetworks. *ACM SIGCOMM 93 Conference*, pages 239 – 248, September 1993. ftp://ftp.cs.colorado.edu/pub/techreports/schwartz.

[DMF97] B. Duska, D. Marwood, and M. Feeley. The Measured Access Characteristics of World-Wide Web Client Proxy Caches. *Technical Report TR-97-16, University of British Columbia*, December 1997.

[Doc] Docspace Home Page. http://www.docspace.com/.

[DP96] A. Dingle and T. Partl. Web Caching Coherence. *Fifth International World Wide Web Conference*, May 6th - 10th, 1996. Paris, France.

[FCA98] L. Fan, P. Cao, and J. Almeida. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *Technical Report, CS-TR-98-1361, University of Wisconsin*, Febuary 1998.

[FZ87] M. Folk and B. Zoellick. File Structures: A Conceptual Toolkit. *Addison-Wesley Publishing Company*, 1987.

[GB93] P. Graham and K. Barker. Distributed Object Base Implementation Using a Single, Shared Address Space. *Proc. Mid-Continent Information Systems Conference*, pages 62 – 77, May 1993.

[GBBZ92] P. Graham, K. Barker, S. Bhar, and M. Zapp. A Paged Distributed Shared Virtual Memory System Supporting Pesistent Objects. *Technical Report TR-92-07, University of Manitoba*, 1992.

[GCR97] S. Gadde, J. Chase, and M. Rabinovich. Directory Structures for Scalable Internet Caches. *Technical Report CS-1997-18, Duke University*, November 1997.

[GRC97] S. Gadde, M. Rabinovich, and J. Chase. Reduce, reuse, recycle: An approach to building large intenet caches. *In Proceedings of Sixth Workshop on Hot Topics in Operating Systems*, pages 93 – 98, May 1997.

[HTM] HyperText Markup Language. *http://www.w3.org/MarkUp/.*

[HTT] HyperText Transfer Protocol. http://www.w3.org/Protocols/HTTP/.

[KAZ+96] P. Keleher, C. Amza, W. Zwaenepoel, A. Cox, and R. Rajamony. Treadmarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18 - 28, 1996.

[LA94] A. Luotonen and K. Altis. World-Wide Web Proxies. *Computer Networks and ISDN Systems*, 27(2):147 - 154, 1994.

[LAJF] B. Liu, G. Abdulla, T. Johnson, and E. Fox. Web Response Time and Proxy Caching. http://www.cs.vt.edu/~chitra/work.html.

[Lit80] W. Litwin. Linear Hashing: A New Tool for File and Table Addressing. *Proceedings of the 6th International Conference on VLDB*, October 1980.

[LNS93] W. Litwin, M. Neimat, and D. Schnerider. LH* - Linear Hashing for Distributed Files. *Proceedings of the 1993 ACM SIGMOD*, May 1993.

[MGB95] J. Mathew, P. Graham, and K. Barker. Object Directory Design Issues for a Distributed Shared Virtual Memory System Supporting Persistent Objects. *Technical Report TR-95-04, University of Manitoba*, July 1995.

[MGB96] J. Mathew, P. Graham, and K. Barker. Object Directory Design for a Fully Distributed Persistent Object System. *Object Oriented Database Systems Symposium of the Engineering Systems Design and Analysis Conference, Montpellier, France*, July 1996.

[OS92] B. Ozden and A. Silberschatz. The Shared Virtual Address Space Model. *Technical Report TR-92-37, University of Washington*, 1992.

[Sar97] C.R. Saravanan. WWW in DSVM. *Master's Thesis, University of Manitoba*, 1997.

[Sch81] M. Scholl. New File Organizations Based on Dynamic Hashing. *ACM Transactions on Database Systems*, 6(1):194 - 211, March 1981.

[SPB98] C.R. Saravanan, R.J. Peters, and K. Barker. WWW in DSM. *IDEAS'98 Symposium, Cardiff, Wales, U.K.*, July 8th - 10th, 1998.

[Squ] Squid Internet Object Cache. http://squid.nlanr.net/Squid.

[TDVK98] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet. *Technical Report, UTCS-TR98-04, The University of Texas at Austin*, April 1998.

[Tre] Treadmarks Home Page. *http://www.cs.rice.edu/ willy/TreadMarks/.*

[Woo96] P. Wooster. Optimizing Response Time, Rather than Hit Rates of WWW Proxy Caches. *Master Thesis, Virginia Tech*, Decembe 1996. http://www.cs.vt.edu/~chitra.work.html.

[Wor94] J. Worrell. Invalidation in Large Scale Network Object Caches. *Master Thesis, University of Colorado, Boulder*, December 1994.

[Yah] Yahoo Home Page. *http://www.yahoo.com/.*