The American University in Cairo

School of Sciences and Engineering

COOPERATIVE WEB CACHING OF DYNAMIC WEB CONTENT

A Thesis Submitted to

The Department of Computer Science and Engineering

in partial fulfillment of the requirements for

the degree of Master of Science

by Mohamed Ramez Mohamed Radwan Atassi

B.Sc., Computer Science

Faculty of Engineering, Alexandria University, 2003

(under the supervision of Dr. Sherif G. Aly)

July / 2010
The American University in Cairo
School of Sciences and Engineering

COOPERATIVE WEB CACHING OF DYNAMIC WEB CONTENT

A Thesis Submitted to
The Department of Computer Science and Engineering
in partial fulfillment of the requirements for
the degree of Master of Science

has been approved by

Dr. ........................
Thesis Committee Chair / Adviser ________________________________
Affiliation ____________________________________________________

Dr. ........................
Thesis Committee Chair / Adviser ________________________________
Affiliation ____________________________________________________

Dr. ........................
Thesis Committee Reader / Examiner ______________________________
Affiliation ____________________________________________________

Dr. ........................
Thesis Committee Reader / Examiner ______________________________
Affiliation ____________________________________________________

Dr. ........................
Thesis Committee Reader / Examiner ______________________________
Affiliation ____________________________________________________

__________________  __________     _____________________    _________
Department Chair/       Date           Dean                      Date
ABSTRACT

Web caching is one of many applications argued to benefit from a switch from a client-server to a peer-to-peer architecture. Several projects suggested the use of a network of peers to provide cached web content in order to help sites survive a burst of user requests. In this thesis, we present a new system that targets web sites with dynamic content and allows them to use a group of a variable number of (volunteered) peers to provide cached web content to the clients.

The main objective of our system is to increase the capacity of a web server, and to reduce the average end user latency. To clarify how our approach would work in a real world web application, we implemented various components of our system in addition to an Internet forums application as a case study of dynamic web sites.

The Internet forums implementation will be used to evaluate our claims. Success will be measured by measuring the average end user latency and upload bandwidth of the web server using our approach, and comparing these values to their counterparts in a normal client-server system.

In our experiments, we used both static and dynamic content. In the first case, all applied requests are view requests, and in the latter, requests include both views and forums replies requests. We varied various parameters, including the rate of requests per second and the number of DHT peers. Results showed that our approach can help overloaded web sites to withstand a high rate of incoming requests while yielding satisfactory end user latency.
# TABLE OF CONTENTS

**Chapter 1 INTRODUCTION** ........................................................................................................... 1

1.1 Overview ................................................................................................................................. 1

1.2 Problem Definition .................................................................................................................. 3

1.3 Thesis Statement ..................................................................................................................... 4

1.4 Thesis Outline ......................................................................................................................... 4

**Chapter 2 RELATED WORK** ...................................................................................................... 6

2.1 Coral ......................................................................................................................................... 6

2.2 Overhaul .................................................................................................................................. 10

2.3 DotSlash ................................................................................................................................... 12

2.4 Squirrel .................................................................................................................................... 14

2.5 Caching of Dynamic Content .................................................................................................. 17

2.6 Other Systems .......................................................................................................................... 18

2.7 Distributed Hash Tables ......................................................................................................... 18

2.7.1 Introducing DHTs .............................................................................................................. 19

2.7.2 Routing and Topology ....................................................................................................... 21

2.7.3 Replication and Data Maintenance .................................................................................... 25

**Chapter 3 APPROACH** .............................................................................................................. 28

3.1 Possible Approaches ............................................................................................................... 28

3.1.1 First Approach .................................................................................................................. 28

3.1.2 Second Approach ............................................................................................................... 31

3.1.3 Third Approach ................................................................................................................ 32

3.1.4 Fourth Approach .............................................................................................................. 33
3.1.5 Fifth Approach (Selected Approach) .................................................. 34
3.2 Web Server ..................................................................................... 36
  3.2.1 Insertion of Files into the DHT .................................................. 38
  3.2.2 Redirection of Web Pages ......................................................... 42
  3.2.3 Discarding Old Data ................................................................. 45
  3.2.4 Incremental Updates ............................................................... 46
  3.2.5 DHT Gateway ........................................................................ 47
3.3 DHT Peers (Volunteers) ................................................................. 48
3.4 Clients ......................................................................................... 50

Chapter 4 IMPLEMENTATION .................................................................. 52
  4.1 The DHT Program ....................................................................... 52
    4.1.1 The Chord/DHash Implementation ....................................... 52
    4.1.2 Adding Support for caching ................................................ 53
    4.1.3 Adding HTTP GET Support ................................................ 54
    4.1.4 Adding Support for Insertions from the Web Site .................. 55
  4.2 The Web Application .................................................................. 55
    4.2.1 The Original Application ..................................................... 56
    4.2.2 Basic Modifications ............................................................. 57
    4.2.3 Inserting into the DHT ......................................................... 58
    4.2.4 Redirecting to the DHT ....................................................... 58
  4.3 The Testing Tool ......................................................................... 60
    4.3.1 Basic Modifications ............................................................. 62
    4.3.2 Modifying the Workload Generator .................................... 62
    4.3.3 Modifying Stats Collector .................................................... 64
Chapter 5 EXPERIMENTING................................................................. 65
5.1 Experimenting Strategy........................................................................ 65
5.2 Setup of the Testing Environment ...................................................... 67
  5.2.1 Simulation of the Overlay Network.............................................. 67
  5.2.2 IP Aliasing .............................................................................. 67
  5.2.3 Traffic Control .......................................................................... 68
  5.2.4 Setting Upload Bandwidth and Delays ....................................... 69
  5.2.5 Measuring the Web Server's Upload Bandwidth ....................... 71
5.3 Experiments ...................................................................................... 72
5.4 Results and Analysis ......................................................................... 75
  5.4.1 Experimenting on Static Content ............................................. 75
  5.4.2 Experimenting on Dynamic Content ....................................... 78
  5.4.3 Varying the Number of Peers ............................................... 80

Chapter 6 CONCLUSION AND FUTURE DIRECTIONS.......................... 84
APPENDIX A Source Code of DHT Program Modifications.................... 87
APPENDIX B Source Code of Testing Tool Modifications ....................... 94
REFERENCES....................................................................................... 109
FIGURES

Figure 2.1 – Coral [3]................................................................................................................8
Figure 2.2 - Overhaul [4]........................................................................................................10
Figure 2.3 – DotSlash [5]..........................................................................................................13
Figure 2.4 – Squirrel: Home Store Approach [6].................................................................16
Figure 2.5 – a Tapestry routing table for a node with ID = 67493 [2].........................22
Figure 2.6 – a Finger Table in Chord and How It Can Be Used to Route Queries [12] .........................................................................................................................................23
Figure 2.7 – a torus for a 2-dimensional coordinate space [1]........................................24
Figure 3.1 – Possible Approach #1.......................................................................................29
Figure 3.2 – Possible Approach #2.....................................................................................31
Figure 3.3 – Possible Approach #3.....................................................................................32
Figure 3.4 – Possible Approach #4.....................................................................................33
Figure 3.5 – Approach #5 (Selected Approach) .................................................................35
Figure 3.6 – Insertion into the DHT when content changes ............................................40
Figure 3.7 – Redirection to DHT when receiving a view request..................................44
Figure 3.8 – Operation of DHT peers ..............................................................................49
Figure 4.1 – Web Site Main Page .......................................................................................56
Figure 5.1 – Latency in Static Experiments, 25KB, using King’s Matrix ....................76
Figure 5.2 - Latency in Static Experiments, 25KB, using Fixed Delays.......................76
Figure 5.3 - Latency in Static Experiments, 50KB file, using Fixed Delays...............77
Figure 5.4 – Latency in Dynamic Experiments, using King’s Matrix .........................79
Figure 5.5 – Latency in Dynamic Experiments, using Fixed Delays .........................79
Figure 5.6 – Latency in a Variable Number of Peers, Static Content.........................80

vii
Figure 5.7 – Latency in a Variable Number of Peers, Dynamic Content .......................... 81
TABLES

Table 3.1 – Example of a Redirection HTML File (Jscript)................................. 42
Table 3.2 – Example of a JavaScript file that creates a web page ....................... 42
Table 4.1 – Messages Table in the Original Web Application ............................. 57
Table 4.2 – Example of a Redirection HTML File (IFRAME) ............................. 59
Table 4.3 – HTTPERF Sample Output .................................................................. 61
Table 5.1. Testing Machine's Specifications .......................................................... 67
Table 5.2 – Creating IP Aliases ............................................................................. 68
Table 5.3 – HTB Queuing Disciplines ................................................................... 69
Table 5.4 – Limiting Traffic When Using the King's Matrix ................................. 71
Table 5.5 – Creating and Examining an IPTABLES rule ...................................... 72
Table 5.6 – HTTPERF Log File ............................................................................ 73
Table 5.7 – Running a DHT peer .......................................................................... 73
Table 5.8 – Running an Experiment using HTTPERF .......................................... 74
Table 5.9 – Server's Average Upload Bandwidth in Static Experiments (Byte / Request) ................................................................. 78
Table A.1 – DHashWeb.C ..................................................................................... 87
Table B.1 – wsesslog.h ......................................................................................... 94
Table B.2 – wsesslog.c ......................................................................................... 94
Table B.3 – sess_stat.c ....................................................................................... 105
Chapter 1

INTRODUCTION

1.1 Overview

Since the introduction of the World Wide Web in 1989, the importance of the web in our lives has been gradually increasing. Today, we have more than 205 million web sites targeting various audiences (according to a survey in February 2009 [29]). Millions of sites are added to the Internet every month.

There are two main steps that are needed to have a site on the Internet. The first is to design the site, and the second is to publish the site on a server connected to the Internet. There are various possible methods for publishing a site. The simplest one is shared hosting, where space on a remote web server is rented. Another option is dedicated hosting, where a remote web server is rented. The third option is possible when a permanent Internet connection is available at the web site owners' premises. In that case the web site can be hosted on a web server in owners' own location.

Over the last few years, fast broadband Internet connections have become cheaper and widely available all over the world. This has encouraged firms and entities with low budget to host their own web sites. Available Internet connection and computers can be used to run small web sites with no additional cost.

However, this solution has two major drawbacks. The first problem is that these connections are usually unreliable. ISPs that provide such connections give no guarantees about up-time and actual speed. The other problem is that the maximum possible upload bandwidth could be too small to even adequately serve a small number of visitors.
In fact, the number of visitors a site can serve depends also on the size of the content uploaded to visitors. Sites that host content of large size such as programs, audio, or video files are under much higher load in term of bandwidth requirements than normal (text + images) sites. One solution that content providers turned to was the use of peer to peer file sharing applications.

Recent years have witnessed an overwhelming increase in the popularity of peer to peer (P2P) file sharing applications. Peer to peer traffic constitutes up to 60% of the total internet traffic according to some surveys [30]. The types of files shared include movies, songs, applications, among others.

Years before this era, similar content was mostly provided by web sites. However, the huge bandwidth requirements of such sites encouraged the move to P2P. Other reasons that favored peer to peer programs are the simplicity of sharing one's own content, and the ability to search shared content of other users of the application.

Peer to peer file sharing applications helped overcome the bandwidth difficulty by letting users upload the content they are downloading to others. Successive improvements on these applications lead to programs with no centralized components and hence more stable networks of peers. Bittorrent used the idea of tic-for-tac to encourage users to share more to achieve higher download speeds [39]. Not only did downloaders of illegal content (like pirated movies and games) benefit from the ability to download large files at high speeds; several software providers - especially software of large size (like Linux distributions) - provide software through Bittorrent. This allows customers/users to download at fast speeds while reducing the required upload bandwidth of providers.
The popularity of P2P file sharing applications had encouraged researchers and programmers to investigate other possible types of P2P applications. These include P2P video broadcasting, P2P instant messaging, P2P multi-player gaming, etc...

1.2 Problem Definition

For many small web sites, the upload bandwidth of the web server that site owners can afford might be inadequate to properly serve the site’s visitors. This would result in degraded quality of service in terms of end user latency and the rate of failures.

Even if the upload bandwidth is sufficient for most of the time, many web sites would face an unexpected burst of a large number of simultaneous users at certain times. One of many possible reasons could be a reference from an article in a new site. This problem is often known as the flash crowds problem.

Researchers suggested various solutions to the mentioned problem. One possible direction is ‘Cooperative Web Caching’, in which several computers cooperate to provide web content to the clients. Many ideas are based on grouping several web sites’ servers, and/or clients, in the form of a peer to peer network.

However, many of the suggested solutions suffer in case of dynamic web content. Dynamic web pages are generated through the use of scripts that form HTML pages with the aid of a site’s database. These pages are usually modified according to user interaction with the site. Examples include pages that enable users to rate contents or to add comments. Some web sites depend entirely on dynamic web pages in order to function, such as Internet forums and E-stores.

Therefore, the need arises for an innovative solution capable of handling the problem of limited upload bandwidth for sites that deliver dynamic content. While in
dynamic sites the processing power of web servers and database servers can become a bottleneck too - especially in large-scale web sites, we focus our attention on the upload bandwidth problem.

1.3 Thesis Statement

This research will propose an approach that enables a web server, that hosts and serves dynamic web content, to increase its serving capabilities by making use of the resources of a group of cooperative (volunteer) computers.

These computers will transparently serve cached web data to end clients (whenever instructed by the web server), thus contributing by processing power, disk storage, and upload bandwidth to the system.

Compared to a single web server configuration, our approach should be able to provide more upload bandwidth, and an overall lower response latency.

1.4 Thesis Outline

This chapter provides an introduction to the problem we are tackling in this thesis. Chapter 2 describes in detail some related research projects in addition to a survey of Distributed Hash Tables (DHTs), which are used by our system to store web content.

Chapter 3 begins by enumerating various possible approaches we could have chosen, and the reasons we didn't. Then we illustrate our approach, including detailed description of all components necessary to build our system.

To verify our claims we will do a partial implementation of our system, in addition to an Internet forums application as a case study of dynamic web sites. Chapter 4 describes that implementation. Chapter 5 describes how we will do
experimentations on the implemented forums application. That includes describing in
details the testing environment. We conclude with the conclusions and future
directions chapter.
Chapter 2

RELATED WORK

The usage of multiple web servers to serve a single web site is a widespread approach used in sites that incur large number of simultaneous web requests. While this approach works well, it is too expensive for small web sites. The successfullness of peer to peer (P2P) programs like Bittorrent and the emerging of distributed hash tables (DHTs) encouraged research efforts trying to use a peer-to-peer architecture to improve the capacities of web servers.

The term “Cooperative Web Caching” is used to refer to systems in which different components of a system (servers or clients) collaborate to provide a cached copy of a requested file to the requesting client. Most of these systems can be categorized into three main approaches:

1- Clients collaborate to form a complete file in an approach close to that of Bittorrent. A client would upload downloaded fragments to other clients.

2- A peer to peer network (organized as a DHT overlay in some projects) is used to act as a cache proxy server between clients and the server.

3- An overloaded web server redirects requests to other content providers. Under low load the server would work normally.

In the following sections, we will examine in depth some collaborative web caching solutions that cover all of the approaches we just mentioned.

2.1 Coral

CoralCDN [3] is a peer-to-peer content distribution network that aims to allow a web site to meet huge demand, using a cheap broadband Internet connection.
CoralCDN is run by a group of volunteer hosts. These hosts act as proxies to the sites subscribed in CoralCDN. Content is replicated as a side effect of users accessing it.

To use CoralCDN, web sites that are subscribed in Coral have to modify their links to web pages. A link is modified by appending “.nyud.net:8090” to the host name in the URL. Through DNS redirection, web clients are transparently redirected to nearby Coral web caches. These caches cooperate to transfer data from nearby peers whenever possible; this results in reduced load on the origin web server and reduced latency experienced by the end users.

To be able to find a cached copy of a web object, CoralCDN uses a key/value indexing infrastructure called Coral. Its job is to map a URL to a list of peers that cache the file located at that URL. Coral is a variation of peer-to-peer distributed hash tables (DHTs). One major difference between Coral and normal DHTs is that Coral allows different values for the same key. For that reason, its indexing abstraction is called a distributed sloppy hash table, or DSHT.

Another attribute of Coral is the distribution of nodes into different hierarchical clusters, where each cluster is characterized by a maximum desired round-trip-time (RTT) between members in the cluster. Coral implementation uses a three-level hierarchy with RTTs of $\infty$, 60 msec, and 20 msec. Queries are first sent to higher level (fast) clusters, and then to lower clusters when no data is found. This method increases the chance of returning values stored at nearby nodes, which results in lower latency.
One major target of CoralCDN is to reduce hot spots that could occur in the peer to peer network. Hot spots are a result of increased popularity of some keys, which results in high traffic targeting nodes that store these keys. Coral uses XOR routing like in Kademlia DHT [27], but with numerous modifications to reduce hot spots. Key/value pairs are stored not only in peers adjacent to the key root node, but are also cached at more distant nodes. In normal XOR querying we try to go as close to the target node (key) as possible. However, in Coral, the distance a query can travel is limited in order to be able to benefit from these distant cached copies. Although this would possibly increase the latency, it would help to reduce the impact of the hot spots problem.

A sample scenario that shows how Coral works is illustrated in Figure 2.1. In steps 1 to 5 a Coral DNS server is contacted and it returns a name server and a Coral proxy close to the user. In steps 6 to 9 the proxy gets the requested file from either a
Coral node or from the origin server. Later on the proxy stores the object and returns it to the client. It also stores a reference in Coral to itself indicating that it now caches that URL.

**Disadvantages**

- The CoralCDN network is still running till now using a large number of volunteer sites. However, replicating the approach would require an infrastructure of strategically located DNS and proxy servers.
- The system relies on trusted servers, if any node in the peer to peer network is malicious, it can easily serve corrupted files to clients.
- The taken approach is not adaptive. Coralized content will always be cached and served through peers even if the origin web server is currently under low load.
- Since all requests to a coralized URL are sent to the peer to peer network, cookies cannot be used to customize the page. No session management is possible if the page is coralized. In other words, a certain URL will always result in the same page for all clients.
- In most dynamic web pages, an expiry date cannot be defined because the content can change at any second because it usually depends on how users interact with the site. However, caching is possible if the dynamic script generates and keeps track of a last modified date (or ETag) for a page. In that case, when a client requests a page that is cached in Coral, the node must always contact the origin web server to make sure that cached page is up to date. The whole coral approach depends on reducing the number of accesses
from the clients to the web servers which makes Coral much less efficient in case of dynamic content.

2.2 Overhaul

Overhaul [4] is a set of changes to the HTTP protocol that promise to help web sites handle flash crowds. It relies on clients cooperating with each other to get a complete web file.

In the proposed system, the web server works normally when it is under low load. When the load increases, the server switches into a mode called Overhaul mode. In that mode, each file served by the server is divided into $n$ chunks. When a request to a web file is received from a client, only one chunk is returned in response.

The response also includes the addresses (IPs/Ports) of other clients that have just requested the file. Also a signature (MD5 hash) of each chunk is included in the response to allow clients to validate chunks downloaded from other clients.

When a client receives the aforementioned response from the server, it begins to contact other clients specified in the response asking for other chunks. In its requests,
a client specifies the list of known peers in addition to the chunks it already has. This enables clients to learn about new peers and chunks that each peer has. The request also indicates the Overhaul port, at which the client listen to requests.

If the peer list is fully exhausted without being able to form the complete document, the client may re-request chunks from known peers. If that fails too, the client may retry the server. The server would reply with a new list of peers, or with the whole document.

Disadvantages:

- Overhaul requires modifications to HTTP protocol at both server and clients. Hence, Overhaul is neither transparent to the clients nor the server.
- The scalability of the system is limited. The server has to maintain a list of peers that are downloading every file served by the server.
- The system would work well if the number of popular files it serves is low, i.e. the ratio of reads per file at any time is high. However if there are many popular files with not so many clients accessing a file at a certain time, a client may not easily find other clients to exchange chunks with.
- NAT and firewalled clients are not supported. A client must be able to listen at a certain port. This is probably the most important disadvantage because it would limit a large proportion of visitors from accessing the site when it is in Overhaul mode.
- Selfish clients can easily cheat the system and download chunks from other peers without uploading any.
- Any page served in Overhaul mode must be identical for all clients. This means that pages cannot be customized according to cookies. Session management is not supported.

- Dynamically generated pages are not supported.

### 2.3 DotSlash

In DotSlash [5], different web servers form a mutual-aid community in which spare capacity is used to relieve web hotspots (flash crowds) experienced by any web site in the community. While the main idea may seem similar to Coral, the approach is quite different. In DotSlash, when a site is under low to normal load, all its content is served using the site’s own web server. When the load increases, the site asks other sites (web servers) for help.

A web server dynamically learns about other servers in the DotSlash community without the need for any administrative intervention. This is accomplished by the use of the Service Location Protocol (SLP) and multiple well known service registries. A web server can use any of these registries to register its information and to search for information about other web servers. Information about a server includes its DNS name and IP, ports used for web and rescue requests, and the maximum allowed redirection rate (which indicates the server readiness to rescue others).

A web server in DotSlash maintains a list of other possible rescue servers to be ready in case of load spikes. A server at any instance is in one of three states: normal state, SOS state if it receives rescue services from other servers, or rescue state if it provides help to other servers. Server cannot rescue others and receive rescue at the same time.
Rescue servers are added and removed as needed (according to server’s load), and web requests are redirected to them either via http redirects or through DNS. In both cases, the operation of DotSlash is transparent to end users (clients).

In case of HTTP redirection, the origin web server returns to the client an HTTP redirection to a URL like “www-vh1.rescue.com”. The vh1 part (virtual host number 1) allows the rescue server to recognize the origin server (a rescue server can provide rescue services to multiple sites at the same time). The client then re-sends the request to the rescue server. If the rescue server finds the file in its cache (cache hit), it returns the file to the client. Otherwise it first retrieves the file from the origin server. The scenario is illustrated in Figure 2.3.

![Figure 2.3 – DotSlash [5]](image)

When DNS round robin is used, a portion of HTTP requests is sent directly to rescue servers by the DNS server. A rescue server recognizes the name of the origin server from the HTTP header (e.g. www.origin.com). The rescue server first checks its local cache for the requested file. In case of a cache miss, the rescue server requests the file from the origin web server, cache it, and then return it to the client. The rescue server requests the file from the origin server using its IP address to make sure it doesn’t get redirected by DNS to another rescue server.
Disadvantages:

- There is no protection from malicious hosts. A malicious server that acts as a rescue server can easily return corrupted files to clients.

- Sites can easily take advantage of the rescue services of other sites without providing any.

- An origin server depends on the availability of rescue servers. The server may not know that a rescue server is down and may keep redirecting requests to it.

- If DNS round robin is used, cookies cannot be used to customize the requested page. In case of HTTP forwarding, cookies can be used only to forward the request to different web pages.

- Increasing the number of rescue servers could increase the upload bandwidth used by the origin web server. The reason is that a certain file would be uploaded to a higher number of rescue servers.

- As in Coral, dynamically generated pages are only supported if the origin web server is cache friendly. Even in this case, DotSlash is inefficient because a rescue server would contact the origin web server (again) for each request to ensure freshness of cached data.

- In case of HTTP redirection, pages are not bookmark-able. The reason is that the bookmark may store a URL that contains the address of a rescue server. The rescue server may stop providing rescuing services to the origin site at later time.

2.4 Squirrel

In contrary to the previously mentioned projects that aim to help web sites deal with high load, Squirrel [6] is directed to the clients. In Squirrel, web browsers on
desktop machines on a LAN share their local caches in order to form an efficient and scalable web cache. This solution intends to replace web cache servers that are usually deployed at the boundaries of corporate networks. The argument is that a web cache server is expensive, in-scalable, requires administration, and represents a single point of failure.

In contrast, a decentralized peer-to-peer web cache like Squirrel pools resources from many desktop machines, and should achieve the functionality and performance of a dedicated web cache without requiring any more hardware than the desktop machines themselves.

Squirrel uses Pastry, a popular Distributed Hash Table (DHT), to locate nodes that cache a copy of a requested object (file). The file URL is used as the key that identify files.

Two different approaches are proposed, implemented, and compared to each other in Squirrel. They are called the Home Store approach and the Directory approach. They differ based on what gets stored in the DHT.

In the Home Store approach (figure 2.4), the browser forwards a URL request to the proxy program installed on the local node. If the requested object is uncacheable, the proxy forwards the request directly to the origin web server. Otherwise, it searches the local cache for a fresh copy of the object, and returns it if it finds one. In case of a cache miss, the peer to peer network is queried using the URL as key. The query will return a node, which is called the home node for that object.
If the home node stores a fresh copy of the object, it will be returned to the client. The home node may also return a not modified message if it finds out that a staled copy at the client is actually still intact. If the home node doesn’t store any copy of the object, it retrieves the object from the origin server, caches it, and then returns it to the client. If the home node has a staled copy, it contacts the origin web server. The origin server would either return a not modified message or a copy of the new object.

The Directory approach is similar to the Home store approach, except the fact that home nodes don’t store copies of cached objects. A home node remembers a small directory of nodes that have recently downloaded a copy of the object. This way, nodes that already have an object in their local caches can serve it to other clients.

**Disadvantages:**

In contrast with previously discussed systems, Squirrel objective is to replace a central cache server in a LAN. So any advantages/disadvantages of Squirrel would have to be in comparison to the central approach. Disadvantages would include:

- Increased latency.
- Increased (aggregated) storage requirements.
- Increased load on desktop machines, in term of processing power and storage.
2.5 Caching of Dynamic Content

In case of large-scale dynamic web sites, the upload bandwidth is usually not the primary concern. Site owners can afford expensive high-speed Internet connections, but the bottleneck is actually in the processing power of the web/application servers, or the database servers.

Edge computing is one of the first techniques that aimed to reduce the load on application servers. Instead of a single server, multiple stateless application servers would be deployed around the globe. These edge servers would access a centralized database server.

Solutions such as Akamai [54] allow page fragments to be cached at edge servers. Their approach entails establishing a template for each dynamically generated page. This template specifies the layout and the content of the page using markup tags [19].

To reduce load on backend database servers, research projects such as MTCache [20] relied on storing a local database on each edge server. The local database would contain materialized views that contain an up-to-date subset of the data in the main database.

All the above mentioned projects target large-scale web applications. In case of cooperative web caching, several researchers targeted dynamic sites. DotSlash approach for dynamic web content [21] relies on caching scripts at rescue servers. These rescue servers would still need to access the centralized database. This approach assumes that the bottleneck is in processing power of the application server.

Globule [22], a collaborative CDN project, proposed two complementary approaches for dynamic content. For applications with high database query locality,
the recommended approach is to cache query results at the replica (rescue) servers. Otherwise, Globule proposes replicating the underlying database at each replica server.

### 2.6 Other Systems

Many other cooperative web caching systems have been proposed by researchers. Most of them incorporate ideas similar to the previously discussed systems. In PROOFS [14], end clients share an unstructured peer to peer network that caches visited pages. CoopNet [15] allows a web server to redirect clients to other clients that have just downloaded the requested file (similar to Overhaul). In BackSlash [16], sites share a distributed hash table. An overloaded server rewrites URLs in documents so that a BackSlash DNS server forwards them to the DHT. Kache [17] is another cooperative web caching system that is based on the Kelips [45] DHT, which allows queries to be resolved in a single hop.

Most of the systems mentioned in this chapter suffer from inefficiency in case of dynamically generated web content, lack for support of session management and page customizations that are usually based on cookies, and vulnerabilities when malicious nodes exist. We will work on handling these issues in our approach.

### 2.7 Distributed Hash Tables

Distributed Hash Tables (DHTs) are used in many of the previously mentioned systems because of their scalability, load balancing, and fast data locating. For these and other reasons, we will use a DHT in our approach. This section provides a background on DHTs.
2.7.1 Introducing DHTs

A Distributed Hash Table is an example of structured peer to peer systems. In order to appreciate the strength of DHTs, it is important to understand the differences between structured and unstructured peer to peer systems. The classification of peer to peer systems into structured and unstructured is based on the presence or absence of structure in routing tables and network topology [11]. In unstructured networks the overlay nodes are organized into a random graph. A joining node would use a random walk starting from a bootstrap node, which is randomly chosen from the set of nodes already in the graph, to find other nodes to fill its neighbors table.

In unstructured systems, there is no restriction on data placement in the overlay topology. Popular data could be replicated at many nodes. Therefore, in order to locate some particular data in unstructured systems, non-deterministic searches are the only option; the nodes have no way of guessing where data may lie. The searching mechanism could be flooding (like in Gnutella [41]). To limit the spread of messages through the network, each query message header contains a time-to-live (TTL) field. The value of the field is decremented at each hop, and when it reaches zero, the message is dropped [2].

Other searching mechanisms were suggested by researchers to improve the performance and efficiency of searching. One example is multiple parallel random walks, where each node chooses a neighbor at random, and propagates the request only to it. Regardless of the searching mechanism used, all unstructured systems give no guarantee that a certain search (query) would be successful. The probability of success depends on the popularity of a queried file.
To remedy some problems in pure unstructured systems, the concept of supernodes has been introduced. A super node is a node that is dynamically assigned the task of servicing a small subpart of the peer network by indexing and caching files contained therein. Peers could be automatically elected to become supernodes if they have sufficient bandwidth and processing power. Supernodes index the files shared by peers connected to them, and proxy search requests on behalf of these peers. All queries are therefore initially directed to supernodes. Although the described system maintains some kind of structure over the network, it is not classified under the umbrella of structured P2P systems [1].

Structured P2P networks employ a globally consistent protocol to ensure that any node can efficiently route a search to some peer that has the desired file, even if the file is extremely rare. Such a guarantee necessitates a more structured pattern of overlay links. By far the most common type of structured P2P networks is the distributed hash table (DHT). There are many DHT implementations available. Examples include Chord [12], Pastry [24], Bamboo [55, 56], and Tapestry [25].

In a DHT, each node is assigned an identifier from a large keyspace. The keyspace may vary from one DHT implementation to another. Chord uses a circular 160-bit ID space. In Pastry nodeIDs are 128 bits in length and can be thought of as a sequence of digits in base 16. A nodeID could be assigned randomly when a node joins the system such that the resulting set of nodeIDs is uniformly distributed across the keyspace.

In distributed hash tables, data is published using a key – as in ordinary hash tables. In many DHT implementations, the keyspace for data keys is the same one used for node ids. The key could be acquired by hashing the data using some hashing
function such as SHA-1. Other options include getting the key by hashing the file name or its path/URL. The DHT network stores a data block in the node whose nodeID is closest to the key (node is called key root). The ‘distance’ function is determined by the DHT implementation. For fault tolerance and durability, the file would be replicated and stored at other nodes as well. These nodes are usually nodes with IDs closest to that of the key root.

When a DHT node tries to find any stored data block (using the data key), a lookup query is sent to a node in the local node’s routing table whose nodeID is closest to the data key. Any node that receives a lookup query will forward it to the next ‘closer’ node until a node that stores the block is reached. This process is the DHT routing scheme which may vary between different DHT implementations.

DHTs were used to implement many types of network applications. Examples include cooperative storage [46, 47, 48], cooperative messaging [49], and event notification [50].

2.7.2 Routing and Topology

In the last few years, many different DHT implementations were suggested. In all of these, nodes take an ID from a large identifier space and organize themselves into an overlay of a specific geometry. There are many types of proposed geometries for DHTs, including trees, rings, butterflies, and others. Each geometry is associated with one or more routing algorithms; and an implementation differs from another by time and space complexity and the degree of flexibility in selecting routes and neighbors. In the next section, we illustrate some of the popular geometries and compare them to each other.
Tree

The Tree geometry is probably the first routing geometry proposed for DHTs. Tapestry [25] is one popular DHT implementation that uses this geometry. In DHTs that use the tree geometry, searching for a node corresponds to doing a longest prefix match of the node ID at each step. Each step takes us deeper in the tree until the target leaf node is reached.

To illustrate how this works, it helps to take a look at how the routing table looks like in Tapestry (figure 2.5). The figure illustrates a possible routing table for a node with nodeID = 67493. The table allows this node to route queries to any other target node using the level 1 entries. However, if the first digit is 3, we can go closer to the target by using the level 2 entries. The longest common id prefix between the target node and the current (intermediate) node allows the route to go even closer to its target. And this prefix matching occurs at each node and allows a query to succeed in $O(\log N)$ hops.

| Entry 0 | 07493 | x0493 | xx093 | xxx0 | x000 |
| Entry 1 | 17493 | x1493 | xx193 | xxx1 | x000 |
| Entry 2 | 27493 | x2493 | xx293 | xxx2 | x000 |
| Entry 3 | 37493 | x3493 | xx393 | xxx3 | x000 |
| Entry 4 | 47493 | x4493 | xx493 | xxx4 | x000 |
| Entry 5 | 57493 | x5493 | xx593 | xxx5 | x000 |
| Entry 6 | 67493 | x6493 | xx693 | xxx6 | x000 |
| Entry 7 | 77493 | xx793 | xxx7 | xxx7 | x000 |
| Entry 8 | 87493 | x8493 | xx893 | xxx8 | x000 |
| Entry 9 | 97493 | x9493 | xx993 | xxx9 | x000 |

Figure 2.5 – a Tapestry routing table for a node with ID = 67493 [2]

The biggest disadvantage in the tree approach is that it doesn’t provide any flexibility in the selection of routes. When routing to a particular target node, only a single entry in the routing table can be used to ensure that we are moving closer to the
target node. On the other hand, the tree geometry has good neighbour selection characteristics. Assuming there are $n$ nodes in the system, a level $i$ entry can be filled using $2^{n-i}$ nodes. Time and space complexity are of $O(\log n)$.

**Ring**

In the ring geometry, nodes are placed on a one-dimensional circle. The distance between two nodes is the numeric identifier difference (clockwise) around the circle. Chord [12, 40], probably the most popular DHT implementation among researchers, is based on the ring geometry. Chord maintains a routing table (sometimes called a finger table) and a successor table of size $O(\log N)$. The successor table contains the addresses of the immediate successors of the node, and can be used to find any node in the system in $O(N)$ hops. To accelerate queries, the routing table is used. The $i^{th}$ entry in the routing table contains a node whose id is in the range $[(x + 2^{i-1}), (x + 2^{i})]$ where $x$ is the id of the current node. Figure 2.6 illustrates how the finger table would help to route queries in $O(\log N)$.

![Figure 2.6 – a Finger Table in Chord and How It Can Be Used to Route Queries [12]](image)

Chord provides good route selection flexibility. To route to any node, the first hop has $\log(n)$ neighbors that progress to the destination and the next node has $\log(n)$ -
1 nodes, and so on. Chord is also very flexible in neighbor selection, we can select from \( n^{\log(n)/2} \) different routing tables for each node [1].

**Hypercube**

The Hypercube routing geometry is based on a \( d \)-dimensional Cartesian coordinate space which resembles a \( d \)-torus (Figure 2.7). The coordinate space is partitioned into an individual set of zones such that each node maintains a separate zone of the coordinate space.

![Figure 2.7 – a torus for a 2-dimensional coordinate space [1]](image)

An example of HyperCubes is the Content Addressable Network (CAN) [26]. CAN differs from other early DHT implementations (Chord, Pastry, and Tapestry) which have \( O(\log n) \) neighbors and hop count; it has \( O(d) \) neighbors and \( O(d \ast n^{1/d}) \) hop count. When CAN’s parameter \( d \) is set to \( \log(n) \), CAN converge to their profile.

In the HyperCube geometry, each coordinate represent a set of bits. It is similar to the Tree geometry because both fix some bits at each hop to reach the destination. However, in the HyperCube, we can modify coordinates (bits) in any order. Because of this, HyperCube has high routing flexibility (\( \log(n)! \) possible routes). However, the Hypercube geometry has poor neighbor selection flexibility. The reason is that each
node in the coordinate space does not have any choice over its neighbours coordinates, since adjacent coordinate zones in the coordinate space cannot change [1].

2.7.3 Replication and Data Maintenance

When inserting a file (data) to the DHT, the routing protocol ensures that the node closest to that file's key (called key’s root) can be found in $O(\log N)$ hops. However, because nodes can arbitrarily fail or leave the P2P network, multiple copies of the same file would be needed to be stored. To be able to retain use of the same routing protocol, DHT implementations normally save these copies at nodes adjacent to the key’s root. Some implementations like Chord/DHash [12, 28] may choose the successors of the key’s root; while others may use some predecessors and some successors.

Location of replicas is one factor that differentiates storage management in different DHT implementations. Other factors include the number of replicas, which could be fixed or belong to a range. Besides saving complete replicas of a file, another technique is used in some DHTs, called erasure coding. In erasure coding a file is divided into fragments according to some algorithm like IDA and a subset of these fragments (of total size equal to file size) could be used to construct the file again.

Whatever the replication method used (simple replication or erasure coding); an important procedure called replica maintenance is run periodically to ensure replicas are in the intended locations and number [42].

Simple Replication vs. Erasure Coding

When using erasure codes, an object is divided into $m$ fragments and then recoded into $n$ fragments, where $n > m$. The rate of encoding is defined as $r$, where $r = n / m < 1$. A rate of encoding $r$ increases the storage cost by a factor of $1 / r$. The key
property of erasure codes is that the original object can be reconstructed from any m fragments [8]. For example, using an \( r = 1 / 4 \) encoding on a block divides the block into \( m = 16 \) fragments and encodes the original \( m \) fragments into \( n = 64 \) fragments; increasing the storage cost by a factor of four. A block would be reconstructed using any 16 fragments.

Many researchers have compared the use of erasure coding and simple (whole-file) replication. Most agree that the use of erasure codes provides more storage savings for the same availability levels (or more availability gains for the same storage levels) than in simple replication. However, there are some disadvantages in erasure coding that may overweight the benefits at least for some applications.

One point against erasure coding is the added complexity in the system. Not only the encoding and decoding are complex operations, data maintenance is more complex than in simple replication because if any fragment is lost (probably because a node left the network) the whole file must be reconstructed in order to create a new fragment. This also yields a more expensive maintenance operation.

Another point against the use of erasure codes is the download latency. When using replication, the data object can be downloaded from the replica that is closest to the client, whereas with coding the download latency is bounded by the distance to the \( m \)th closest replica [13].

Erasure coding is also less efficient in situations like when it would be required to download a subset of the stored file or when operations in a system is done at server level, like in keyword searching.
Placement and number of Replicas

Since the introduction of DHTs, intensive research efforts have been spent trying to increase the availability and durability of data in DHTs. The most important limiting factors are the amount of increased (redundant) storage to store replicas and the bandwidth required to synchronize and maintain these replicas. First DHT implementations used a fixed number of whole-file replicas which were stored on nodes adjacent to key’s root. Data maintenance procedure was run periodically in which adjacent nodes synchronize with each other to ensure replicas are in the right place and in the right number.

Several enhancements were suggested by researchers. In addition to erasure coding which was described earlier, many research papers argued that the use of a range for the number of replicas instead of a fixed number help to decrease bandwidth requirements for data maintenance [7, 10]. Researchers suggested different ideas for placing replicas [23]. Some researchers suggested storing replicas at random nodes and store pointers at key’s root and adjacent nodes. When it is better to synchronize replicas is another open research question, some call for a proactive approach where others call for a lazy one [9].
Chapter 3

APPROACH

Before explaining our selected approach, we begin this chapter by examining other possible approaches that we could have taken, and the disadvantages associated with them. Later sections discuss in details our approach and all the components that constitute our system.

3.1 Possible Approaches

We investigated many different approaches to tackle the defined problem. Most of these approaches differ from each other according to the order in which messages flow between clients, peers (volunteers), and the web server. Some approaches are based on ideas seen in some of the related projects we previously discussed. In this section, we discuss these possible approaches, and the disadvantages that prohibited us from adopting them.

3.1.1 First Approach

In the first approach we investigated (Fig 3.1) we tried to find out if it is possible to rely solely on the peers to run the basic operations in a web application. If this is possible, we could significantly reduce the load on the web server. That will also lead to increasing the availability of the system, since most operations don’t rely on the web server.

Considering the case of static sites, the solution is relatively simple; the site would have to push web files whenever they get updated to peers. The main problem here is how clients would be able to transparently browse a site through the peers. One possible method is to use a DNS server that returns an address of a random peer,
which would return the file at the requested URL if it stores it. Otherwise, it will redirect the client to another peer that stores the file at the requested URL.

**Figure 3.1 – Possible Approach #1**

In case of dynamic sites, the most significant problem is that there is no single database as the one that usually exists in normal web applications. Data is scattered between all peers in the system; which makes querying and updating the data difficult. To illustrate some of the difficulties, one can consider a simple dynamic web site: a blog site which allows visitors to add comments to blog entries.

In order to maintain availability, all pages are replicated at several peers of the P2P network, which takes the form of a distributed hash table. In this proposed architecture, the peers must be responsible for the addition of comments to pages; the server should not be contacted.

The first difficulty comes from the fact that the set of peers that have replicas of a comments file is dynamic; peers could join and leave at any time. The problem is further complicated because concurrent updates could occur. The case was different in
static sites because there is only a single uploader/updater (the web server); a version field would be used to ensure consistency in that case.

Another major difficulty in updating is the possibility of network partitions. Consider the AUC’s own network. If the external link to the Internet is broken, and there are several DHT peers active inside the internal network, these peers can connect to each other and form their own DHT. If a page file is stored at one of the nodes inside the AUC DHT, it can be updated in the same time by different users at both the outer DHT and the internal DHT; which results in an inconsistent file. Brewer’s conjecture states that a web service cannot be available, consistent, and immune to network partitions at the same time [24]. We believe that this conjecture is true in the case of our distributed system.

In our simple application, a normal requirement would be that comments are associated with a date/time field, which indicates when the comment was added. Since we have no central server involved when a comment is added, using the local time at the updater PC would lead to a page in which comments’ dates are not in the same order as the order of the comments themselves.

Other factors would also lead to increasing the difficulty of this approach, one example is the existence of malicious peers. These peers must not be able to delete blog entries or comments. Increasing the complexity of the application, for example by requiring clients to authenticate, would cause more difficulties in our approach, especially if peers can be malicious.

Because our main objective is to increase the capacity of the web server, and because updates are often much less than view requests, we decided to discard this approach and keep the web server responsible for all updates to data.
3.1.2 Second Approach

In the second possible approach (Fig 3.2), clients still contact the peers first. Peers store cached copies of web pages. However, peers will need to contact the server only if it receives a request for a file that is not cached in the peer to peer network. Also, clients would need to contact the server directly in case of updates.

![Diagram showing network flow](image)

Figure 3.2 – Possible Approach #2

This approach is similar to the one taken by CoralCDN (Section 2.1). While this approach does decrease the load on the server, it suffers in the case of dynamic content. In dynamic content the server would always be contacted because we cannot predict when the content was changed. Nonetheless, the main problem is that all requests for any page using a URL will always give the same result. This means that pages cannot be customized according to the visitor, which could be done through cookies. Also, any small modification to a page would stale the cached copies of it. These reasons are enough to discard this approach.
3.1.3 Third Approach

Some of the problems described in the previous approach were caused by the fact that clients contact peers first. So beginning from this approach, we would like our clients to contact the server first, which would delegate the request to peers if needed.

In this approach, we investigate if it is possible for the server to delegate the client’s request directly to the peers, without passing by the client in the midway. One of the peers would have to return the requested file to the client. Figure 3.3 illustrates the described scenario.

```
Client                  Web Server
                   1
                 3  2
     P2P Network
```

Figure 3.3 – Possible Approach #3

This approach would be possible only if a proxy application is installed at the client. The application will forward the requests it receives from the browser to the server, and return a response only when a peer replies (or after a timeout). While using this approach means that the client transparency objective will not be met, the real problem lies in the fact that NAT and firewalled clients are not supported. The client
must be able to listen on a fixed port to be able to receive responses from peers. Many end users will not be able to meet this requirement.

### 3.1.4 Fourth Approach

In this approach, a client sends the request to the server, and the server redirects the client to one of the peers in the peer to peer network. The peer returns a page if it caches it, otherwise it searches for it in the peer to peer network. If the page is not in the P2P network, the peer would retrieve the file from the web server.

![Diagram](image)

**Figure 3.4 – Possible Approach #4**

Figure 3.4 illustrates this approach. Note here that the redirection is done through a standard HTTP redirection response. In this approach, cookies can be forwarded to the server. The server could use values in a cookie to redirect the visitor to one of different possible pages. Also, further customization is possible by passing values to the peer by URL parameters. A JavaScript in the page can customize it based on these values.
A problem we would face with dynamic content is that the peer must contact the web server to ensure that the cached page is up-to-date. To solve this problem, the ETag of the page that specifies the last version of it can be sent to the peer via a URL parameter.

One problem with this approach is that URL parameters cannot be used to identify a web page. The key that identifies a page in the P2P network must be a URL without its parameters. This is mandatory because we need to use the URL parameters to pass values to the peer and the page.

Also, in this approach, pages are not bookmark-able. The reason is that the bookmark will contain the address of the node, which could go offline or unsubscribe of the system at any time.

3.1.5 Fifth Approach (Selected Approach)

In our selected approach, we begin from the previous (fourth) approach and try to modify how it works to solve the disadvantages associated with it.

First, instead of redirecting clients using a normal HTTP redirection message, the web server (application) returns a small HTML page (Figure 3.5). This page would retrieve web content from an external source (a peer in the P2P network) and embed it in the page. This is possible through the use of a JavaScript tag in the HTML page, with the source property of the tag pointing to the external source.

The actual JavaScript stored in the external source would recreate the original web page, using a function such as "Document.Write". This way, the address bar in the browser would still point to the origin web server. This means that pages are bookmark-able and any refresh would go through the web server again. Also, this enables customizations to the page by defining JavaScript variables in the HTML file.
and consuming them in the JavaScript file store in the DHT. One possible example is passing the name of the logged in user to the JavaScript file in the DHT.

The external source we mentioned is in fact a peer in a peer to peer network, which takes the form of a distributed hash table (DHT). This means that the web server must be able to track peers in the DHT to be able to forward clients to peers. This can be accomplished by joining the web server into the DHT.

The second difference in our approach comes from the need to protect the data from being corrupted by malicious peers. We want both clients and peers to be able to detect any invalid data stored in and retrieved from the distributed hash table. To accomplish that, files are stored in the DHT using their content hash as a key. Hence, when the web server redirects clients into the DHT, it forwards them using the hashing value as a key instead of a URL (as most approaches do). This means that the web server must be aware of the content of the stored file and the file's key before the redirection.
The third main difference in our approach from others is in the method that dictates how cached data is inserted into the P2P network. In all the related projects we examined in Chapter 2 (like Coral and DotSlash), caching of files is done using a cache-on-access rule. Using this rule, content is cached when a user requests it. So according to this rule, when a peer receives the client's request after it has been redirected by the web server, the peer would search first the DHT and then request the file from the web server in case of a cache miss.

However, our approach uses a push model (in contrast with the pull model just described). The web server would be responsible for pushing content to the DHT. How and when the pushing occurs and why did we choose this model is explained in the next section, which elaborate on the responsibilities of the web server. Since the web server is responsible for pushing files into the DHT it keeps track of the files in the DHT and would only redirects clients to the DHT if the requested file is stored there.

In the next three sections, we examine in details the three main components of our system – the web server, the DHT peers, and the clients.

3.2 Web Server

When the term "Web Server" is mentioned in literature, it could refer to the physical computer that runs a web server program like Apache or to the web server program itself. We use the term Web Server to refer to the actual machine – which is a node in our distributed system – including the web server program (Apache) and the web application hosted on it. However, most of our added functionalities are implemented as part of the web application hosted on the web server.
The web server is responsible for many tasks in our system; the most important two tasks are the insertion of web files into the DHT and the redirection of clients into the DHT peers. We will explain these tasks in details in the next sub-sections.

Our web server works normally when it's under low load. It neither redirects clients to the DHT peers nor inserts files into the DHT. It returns the actual HTML files as if our system is not installed. Web server programs like Apache provide information we can use to determine if the current load is considered high. This includes the current rate of incoming connections, the number of concurrent requests being processed, and the average time taken to respond to requests.

There are multiple reasons for turning off our system when the load is low. The most important one is that, in the case of low load, the latency experienced by clients would be better without using our approach. The second reason is to reduce unnecessary load on our volunteered peers, since we can handle the entire clients' load ourselves.

When the load on the server increases, the web server starts redirecting clients into the DHT peers. The process of redirection is as we described in the previous section, the web server returns a small HTML file that consists mainly of a JavaScript tag that points to a DHT address (peer IP/port and a DHT key). The JavaScript file stored in the DHT would reconstruct the original HTML page.

Since the web server is responsible for the insertion of web content into the DHT using a push model, the server is aware of all files (and their versions) that are stored in the DHT. Hence, it only forwards requests for files stored there. A more complex implementation would redirect only a portion of the requests to the DHT, depending on how high the load on the server is.
3.2.1 Insertion of Files into the DHT

In centralized cache servers and in all cooperative web caching projects we reviewed, a pull model (cache-on-access) approach is used to fill the caches with web sites' files. In this model, files are cached as a result of clients’ requests. Whenever a client requests a file, it gets stored in the cache. When the cache is full, least recently accessed files are removed.

Instead of the pull model, our approach uses a push one. The server is responsible for pushing files that would be served by peers to the clients into the peer to peer network. The server can either push all files, or push a subset of the files based on some defined criteria like popularity and probability of modification. Files that should not be visible to public should not be inserted into the DHT. Also, pages secured by SSL should not be cached. In all cases, files are inserted into the DHT using their content hashing value as a key, to allow peers and clients to validate any downloaded file.

In static web sites, where each document is a simple HTML file created by an author, the HTML files could be uploaded through a script in the site that stores the URL of the file along with the content hash (the key) in a local database. Access to files is enabled through a script that retrieves the key of the requested file from the database and then forwards the request to a DHT peer.

On the other hand, in dynamic web sites, updates on a site's contents occur more frequently depending on users' interaction with the site. In these sites, we have several alternative methods for insertion into the DHT. Insertion into the DHT can be implemented either by a periodic process, or as a result of some event, such as a page creation, view request or update.
Similar behavior of cache-on-access method can be achieved by pushing all files when they are requested for the first time, and then redirect the client's request to the newly inserted file. As in static sites, whenever a file is inserted into the DHT, the server records the ID of the page, its DHT key, version, and time-to-live in the site's local database.

In our implementation that is based on an Internet forums application (discussed in the next chapter), we choose to push any file whenever it gets updated. In other words, we push files whenever a client creates a new thread or reply to an existing one.

A major problem of this approach is that we have to decide either to delay some requests, like the first view request (or modification requests in our forums application), until the requested file is inserted into the DHT or lose some bandwidth and return the full page from the web server until the file is in the DHT. In the second case, later requests for the file (if any) would access the file from the DHT. However, in that case we cannot be sure that there would be any later request to the same file.

To ensure that we do not waste bandwidth, we choose the first option, albeit its negative effect on the latency experienced by some of the clients' requests. We benefit from the fact that in our forums application and many of similar ones, after all data updates requests the web application would redirect the client to the newly updated page. Hence, when we insert a page into the DHT after a data update we can delay the redirection to the viewing page until the file is inserted into the DHT.

Figure 3.6 shows the steps taken when a web file gets modified. Since we cache files whenever they get modified, "Cache on Change Rule" always returns yes. In the
figure we assume that we do not differentiate between files in terms of attributes such as popularity; all files are cached when modified.

The other disadvantage of the used push model is that when we save actual files into the DHT instead of links to cached-on-access copies, we use more bandwidth from the DHT peers. The reason is that we create a fixed number of replicas when a file is inserted into the DHT (three in our implementation). This could be a problem if the number of peers in the DHT is low and the rate of updates to files in the target web application is high compared to reads.

While these two disadvantages in the push model we selected are important, there are also advantages that lead to the selection of that model. That selection is motivated by the objective to decrease the upload bandwidth of the web server as much as possible.

If we first compared our approach to an approach similar to DotSlash where cache servers are not connected to each others, we find out that every cache server
would need to get a copy of a file from the web server to cache it. This means that any increase in the number of cache servers could increase the load on the web server.

In another possible approach, a DHT could be used to store links to cached copies of a file. When a file is requested for the first time, the peer receiving the request would not find a reference to the file in the DHT and hence request it from the web server. When another peer receives a request for the same file, it gets it from the first peer (after querying the DHT), caches it, and stores a link to itself in the DHT (using the file's key).

In the described approach, a single file could get requested from the web server more than once. This is possible if several requests to the file happen at the same time when the file has not been cached yet. The peers that receive these requests will find nothing in the DHT and all will request the file from the web server.

Another scenario is possible when a single peer receives the first request to the file. If the peer goes offline or become heavily overloaded, subsequent requests to the file would try to access that peer and fail. Hence, they would have to get the file from the web server. On the other hand, when our approach is used, the probability of uploading a certain file from the web server more than once is very low.

In many dynamic web applications, a newer version of the same page would share a lot with the previous version. In some of these applications, like our implemented Internet forums application, the newer version can be constructed from the older one using a simple JavaScript. This feature is discussed in detail in subsection 3.2.4 (Incremental Updates). It helps to decrease the used upload bandwidth of the web server. It benefits from using the push model because the web server already knows the versions of pages currently stored in the DHT.
3.2.2 Redirection of Web Pages

The method of forwarding is as has been specified previously in the approach summary; when the load on the server is low, the server will return the full HTML page. When the load is high and the file is stored in the DHT, the server will return a small HTML file, with an embedded JavaScript tag that its source tag points to a DHT location. The JavaScript file that is stored in the DHT would contain (at least) a single instruction that creates the original web page, as would it look like if it has been retrieved directly from the server.

The web server can find out if the requested file is cached in the DHT through the local database. Figure 3.7 shows the steps taken by the web server when a view request arrives. Table 3.1 shows an example of a redirection HTML file, where the third line contains the redirection JavaScript tag. This HTML file is further explained later in section 3.4 (Clients). Table 3.2 shows a simple JavaScript file that creates a "Hello World!" web page. The file stored in the DHT would use a similar method to construct the original web page.

<table>
<thead>
<tr>
<th>Table 3.1 – Example of a Redirection HTML File (Jscript)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;script&gt; var username=&quot;John&quot;;&lt;/script&gt;</code></td>
</tr>
<tr>
<td><code>&lt;script src='www.mysite.com/libV1.js'&gt;&lt;/script&gt;</code></td>
</tr>
<tr>
<td><code>&lt;script src=DHT('1234567890123456789012'))-&gt;&lt;/script&gt;</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3.2 – Example of a JavaScript file that creates a web page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Document.Write('&lt;html&gt;&lt;body&gt;Hello World!&lt;/body&gt;&lt;/html&gt;');</code></td>
</tr>
</tbody>
</table>

To reduce the load on the server, embedded links to immutable files in a web page that are stored in DHT could point directly to the DHT. Immutable files refer to files that are not expected to be modified, like images. By pointing links directly to the
DHT, the web server doesn’t have to bother redirecting clients to images, thus reducing the load on server. This also reduces the end user’s latency, since the clients access the DHT directly without passing by the server. Other files used throughout the site like CSS (Cascade Style Sheets) files and general (library) JavaScript files are usually cached by a user’s browser. Hence, there is little benefit in storing them on the DHT.

However, embedded links to web pages don’t follow this rule. Links to web pages in a web page that is stored in the DHT (or returned from the web server) are always normal http links that points directly to the web server. There are many reasons behind this rule:

- It ensures that data is up to date. The DHT data cannot be updated because content hash is used, a newer version of a page would be written to a new DHT block with the new key. This means that the client can refresh a page, and the server would redirect him/her to the latest version of the page.

- Bookmark-ability. Clients can bookmark any web page because the page's address points to the web server not to a peer.

- Passing by the web server allows it to record hit counts (number of visits to a certain web page).

- The web server must be contacted whenever there is post data, for example, when a user adds a new comment to a page.

- The web server is able to customize the returned web page and manage sessions. Also minor changes in the page can be applied without invalidating the cached page by using JavaScript variables.
Customizations are possible through the small redirection HTML file returned by the web server. This file can customize the page returned to the user, by passing values via JavaScript variables to the script stored in the DHT. Since the server is contacted first, cookies can be used to track sessions, and to identify the current logged in user to a site. The JavaScript variables could customize the page according to that user, such as by displaying the name of that user.

**Figure 3.7 – Redirection to DHT when receiving a view request**

Another usage of JavaScript variables is to reduce the chance a file stored in the DHT becomes outdated. In a forums application like the one we implement in the next chapter, every thread page displays the number of pages in the threads and links to them. If this value and links were hardcoded into the page, then when a new page is added to a thread, all other pages get invalidated.

However, we can pass the number of pages using a JavaScript variable to the JavaScript files stored in the DHT. These files can show the number of pages and construct links to pages programmatically using JavaScript functions. Other methods to customize the page using JavaScript are explained later in this chapter.
3.2.3 Discarding Old Data

One limitation of DHTs is the absence of deleting capability. When a data block is inserted, a time-to-live (TTL) value is used to specify when the block should be discarded. We would need deletion for two reasons. The first one is to remove old web pages that are no longer popular and consume unnecessary disk space and bandwidth. The other reason is to remove outdated data. Since content hashing is used, when any modification has to be applied to a file stored in DHT, we have to reinsert the new file with the new key and discard the old one. Since in dynamic sites updates could be unpredictable and with high rates, outdated data is a major problem. Outdated data consumes not only disk space, but also bandwidth which would be needed to synchronize replicas.

A possible solution to this problem is to start with small time-to-live (TTL) values, and then repeatedly use higher values when renewing the TTL (up to a certain threshold). In other words, we would have to track DHT files and renew them just before they expire. And each time we renew a file, we would increase the time-to-live, except if the file popularity has decreased.

The web server is responsible for maintaining the files inserted into DHT along with their expiry dates. It's responsible for renewing the files in the DHT before they expire, or delete them (or mark them as invalid) from its local table that tracks DHT files. However, since in our experiments we apply workloads on the web application for a very short period of time, this problem is irrelative to our experiments. We will not implement expiry tracking and hence will not elaborate more on this problem.
3.2.4 Incremental Updates

In many dynamic web applications, an updated page could be constructed using a small modification to the previous version. Consider a blog site that enables visitors to comment. Any new comment would invalidate the page; but it’s possible to recreate the new page using a simple JavaScript that adds the new comment. To implement this idea in our system, the web server could return an HTML file that contains a reference to the DHT, in addition to a JavaScript that adds the new comment.

If the JavaScript file becomes large, probably because of other added comments, the whole page could be reinserted to DHT. Another possible option is to insert the new JavaScript file into the DHT; and in this case the HTML file returned by the server would contain two or more references to the DHT. We call this feature ‘Incremental Updates’.

In other types of sites where portions of the page could be changed independently, and in sites where the layout can be changed according to the user, we can choose to divide the page into various fragments. Fragments can be glued together using the JavaScript in the HTML file returned by the web server. When the server is returning the whole page directly to the client (low load), fragments should not be used because they would increase the load on the server.

The main reason for using this feature is to reduce the upload bandwidth used by the server. Instead of uploading an updated version fully to the DHT, the server could upload the difference between the new and the old version to the DHT. Requests to the page would receive a redirection HTML file that contains references to the two DHT files. However, notice that this operation lead to slightly increase the size of the redirection HTML. If the number of read requests is very high compared to the
number and size of modifications, this small difference in the size of the HTML file could out-weight the benefit of using the incremental JavaScript.

While our approach and push model enable using this feature, we will not implement nor test it. Using this feature requires careful analysis of the target application and comparing the benefits versus the downsides of using it.

3.2.5 DHT Gateway

One of the tasks that could be assigned to the web server is to act as a gateway for the DHT for both DHT peers and clients. While any peer that is always online can act as a gateway for the DHT for DHT peers, the problem is more complicated in case of the redirection of clients to peers. We talk more about this problem later on when discussing the clients at the end of this chapter.

If the web server is responsible for returning an address of a DHT peer for a client, it must be able to get the address of a random peer in the DHT. This is possible by joining the web server into the DHT network. However, this approach has the disadvantage that the web server becomes responsible for storing DHT content and returning them to the clients.

Another option is to modify the DHT implementation and allow the web server to be part of only the routing layer and not the DHT layer. However, this approach is complicated. Another far simpler option is possible if one (or more) of the DHT peers is always on and has a static IP. The web server can contact that peer at certain time intervals and get a list of random DHT peers. In our implementation we choose the default scenario where the web server joins the DHT network.
3.3 DHT Peers (Volunteers)

Our system depends on a group of volunteer computers that contribute bandwidth and disk space to the system, by joining the DHT network. These volunteers could be site’s owners, dedicated site members, or web servers of other sites that form a collaborative community with our site.

Since DHTs are inherently scalable the number of peers that can join the network is virtually unlimited. On the other hand, the system can work and benefit from the existence of only a small number of peers.

These peers would have to run a DHT program (we use Chord/DHash in our implementation). Each volunteer would typically represent a single DHT node in the peer to peer overlay. A peer can join and leave the P2P network at any time. However, the rate of the node joining/leaving (churn rate) is assumed to be small in comparison to many other P2P applications.

Volunteers could be either trusted or un-trusted. The latter case means that some of the volunteers could be malicious. We are mainly interested in protecting the content of the web pages provided by the site. Because of that, if malicious peers could exist, the clients would have to verify the hashing value of any file served through the DHT.

Since volunteers are responsible for serving web content to clients (after clients get redirected by the web server), the DHT program must listen on a fixed port for HTTP requests from clients. The program only recognizes a specific pattern of HTTP GET requests that specifies a hashing value (requested file's key) in addition to other optional parameters.
When receiving an HTTP request, the DHT program searches the local database using the hashing value as the key. If the file is found the program returns it to the client. If the file is not found, two approaches are possible. In the first one, the program initiates a DHT query, and then returns a redirection to a node that stores the required file.

In the second approach, the program retrieves the actual file from the DHT, caches it, and then returns the cached copy to the client. To decrease the client's latency, especially in large files, the program can send the file to the client as it being downloaded from the other DHT peer. A mix of both approaches is possible; the program can forward requests if it's under high load, or use the second approach otherwise (Fig. 3.8). We use the second approach in our implemented application.

Figure 3.8 – Operation of DHT peers
3.4 Clients

In our system, clients should be able to retrieve files from the DHT network. In our attempt to achieve that, we should try to meet an important objective, which is client transparency. If our system is transparent to clients, site visitors won’t have to install any program or browser extension for the system to work.

To download files from the DHT, the client should connect to a random peer in the DHT. This peer performs queries on behalf of the client, and returns either the file or a redirection to another peer that stores the actual file. This method requires the client to be able to retrieve the address of a random peer. Several methods could be used to accomplish this.

One method is to use a DNS server hosted by any peer in the DHT, which could be the web server. The peer would periodically retrieve a set of random peers from the DHT, by querying some random keys. When a DNS query arrives at the DNS server, the server returns one of the random peers. The reply also must indicate a short time-to-live (TTL), which should typically be less than a minute.

The DNS approach might not be favored because of the overhead in maintaining your own DNS server. Another important reason is that many DNS servers and resolvers ignore the TTL value set by a domain and override it by their own. For example Internet Explorer would cache a resolved DNS for thirty minutes by default. If the peer retrieved through DNS goes offline, the client would have to wait for a long time until an address of another peer is retrieved.

Instead of the DNS approach, the client can retrieve the address of the random peer through an AJAX function call to the web server (using JavaScript). Another function will be used in all links to DHT, to construct an http link using the random
peer address and the DHT key. Table 3.1 shows an example of how the HTML files returned from the server would look like. The DHT function is responsible for concatenating the peer address with the requested key.

The last random peer address returned from the server will be stored in the site’s cookie along with the time of retrieval. This address will be used for next calls to the function until a predefined time-to-live passes. This will reduce the load on the server and the latency experienced by the end client.

One problem of the previously described approach is that it’s not immune to malicious DHT peers. If the existence of such peers is possible, we would want from our clients to check the hashing value of any file downloaded from the DHT. In principle, it is possible to read a file and calculate a hashing value from its contents using JavaScript. However, a problem appears when we try to read files from a domain other than the domain of the original page. Most browsers disallow all cross-domain references for security reasons. This means that for some browsers where there is no work around for this problem, the clients may have to use a browser’s extension if some DHT peers could be malicious.
Chapter 4

IMPLEMENTATION

To illustrate our approach and its applicability to real world web applications, we will implement various components of our system in addition to an Internet forums web application. The implemented system will be tested to evaluate the success of our approach. The next chapter will explain how we tested the implemented system on a single machine by simulation. This chapter illustrates in details how we implemented the various components of our system. This includes a description of the modifications we did to the testing tool that we will use in the next chapter.

4.1 The DHT Program

The DHT program is responsible for storing cached copies of web files in the peer to peer network and delivering them to the clients. We will examine the DHT implementation we used and how we modified it.

4.1.1 The Chord/DHash Implementation

For our DHT peers we used the Chord/DHash research implementation. Chord/DHash is one of the most popular DHT implementations, especially among researchers. Chord refers to the routing (lookup) layer while DHash refers to the DHT (storage) layer. Chord/DHash is based on the ring topology, and thus it possesses many of the advantages of this topology such as the flexibility in the selection of routes and neighbours.

Chord/DHash is implemented using C++ and the SFS libraries. SFS libraries provide many useful components for simplifying the development of event-driven network programs in C++. Many of these components are used in the Chord
implementation, enabling features such as: asynchronous networking programming, asynchronous RPC, callback programming (events), reference counted objects, and many data structures such as dynamic arrays and hash tables.

We will run the Chord/DHash implementation keeping most of the default customizable parameters. This includes using simple replication and three replicas per DHT block. For the routing protocol, we will use Accordion [33]. Accordion automatically modifies the size of the Chord routing table to improve the lookup performance. In case of small networks, lookups can be achieved in O(1) instead of O(log N).

4.1.2 Adding Support for caching

To run a Chord/DHash node, three processes are needed. LSD (location service daemon) is the main process, ADBD is the database daemon and Maintd is the DHT maintenance daemon. We will add another process called DHashWeb, which will be responsible for receiving DHT fetch requests from the clients and DHT insert requests from the site. DHashWeb listens on a single TCP port specified by a command line parameter.

DHashWeb connects to the DHash Unix socket created by the LSD process. We use this socket to issue the two basic DHT commands to our DHT, which are "insert" and "retrieve". DHashWeb also connects to the ADBD Unix socket in order to cache fetched web files in the local database.

We previously defined various possible approaches for DHT peers to return data to clients. In all cases, the DHT peer will return the requested file if it is responsible for storing a replica for it. If it's not, the peer would search for peers that do store the
file. Here we have two possibilities, redirect the client to that peer, or get the file from that peer and send it to the client.

In our implementation, we implemented only the latter approach. And since peers get files before sending them to clients, it makes sense to cache these files and use cached copies for later requests. The original DHash implementation does not support caching, and thus we have to connect to the local database from the DHashWeb process to be able to store and retrieve cached files.

When a request for a file arrives at DHashWeb through the TCP port (more on that in the next section), the first step is to check the local database, which contains cached files and files that the local DHT peer is responsible for. If the file does not exist then a retrieve request is issued to get the file from the DHT. After the file is retrieved, it get stored in the local database, and sent to the client. One possible optimization to this technique (not implemented) is to send the file to the client during the retrieval process from the DHT. This would decrease the latency experienced by the client in case of large files.

4.1.3 Adding HTTP GET Support

To simplify the implementation of the client component, the DHT peer listens to standard HTTP GET requests from the clients. When the DHT peer recognizes that a TCP connection is established (could be from a client or the site), it reads data as soon as it arrives through the connected socket. If the first received word is GET, then this is an HTTP GET request. The second word in that case would be the requested URL. Other data sent can be safely discarded.

Our program would only recognizes URLs that begin with the phrase "/gethtml?id=", followed by a key which is a string of hexadecimal characters that
specify the requested file. Any other pattern of URLs would result in an error returned to the client. The file would then be retrieved as described in the previous section and sent to the client. Then the connection will be closed by the peer.

But before sending the contents of the file, the peer would have to send the HTTP return code 200 which means OK through the socket. Also, the type of the content would need to be specified. Since in all of our experiments we will store only html files in the DHT, we will always return text/html as the content-type. However, in a real world implementation, files like images would be stored in the DHT. Hence we would need to either store the type of the file as auxiliary data in the DHT, or specify the type of the file in the URL.

4.1.4 Adding Support for Insertions from the Web Site

The site needs the ability to store files to the DHT. It is easier in this case to forget about the HTTP standards, and send the file using a custom 'protocol' through the TCP socket. Our site, which will be implemented using PHP as will be seen later in this chapter, can easily create a TCP socket and use it to connect to a DHT peer and send the file to that peer.

The word 'STORE' is first sent through the socket followed by the length of the file and then the actual file. The peer would store the file in the DHT and in its local cache and then return the key of the file to the web site.

4.2 The Web Application

We ran our experiments on an Internet forums web application (site). We used an open source implementation and modified it for our purposes as described below. The forums application we used was retrieved from [36]; it is an open-source PHP application that uses a MYSQL database.
4.2.1 The Original Application

The site allows its visitors to discuss any topic. Any visitor can create a thread or view/reply to existing threads. The site main page lists all the threads in the site (Figure 4.1). Each thread contains an original message and a list of replies. Replies are divided into pages of twenty replies per page. Each post (original message or reply) consists of a title, author, creation date, and a message.

![Forum Main Page](image)

**Figure 4.1 – Web Site Main Page**

There are three main PHP pages in the site; these pages were modified to support our approach. The three pages are the "newthread.php", "reply.php", and "viewthread.php".

The MYSQL database used in the original application consists of a single table called messages. Columns include an auto-increment field called 'message_id' as a primary key and a nullable column called 'message_parent_id' as a foreign key to the same table. The list of threads (main page) displays all rows (messages) with the parent field equal to null. The columns of the messages table are shown in Table 4.1.
### Table 4.1 – Messages Table in the Original Web Application

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>message_id</td>
<td>int</td>
<td>auto_increment – primary key</td>
</tr>
<tr>
<td>message_parent_id</td>
<td>int</td>
<td>Foreign key</td>
</tr>
<tr>
<td>topic</td>
<td>varchar(50)</td>
<td></td>
</tr>
<tr>
<td>author</td>
<td>varchar(50)</td>
<td></td>
</tr>
<tr>
<td>message</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>date_entered</td>
<td>datetime</td>
<td></td>
</tr>
</tbody>
</table>

#### 4.2.2 Basic Modifications

Several modifications were needed to be applied to the downloaded web application. Since we may choose to insert only a subset of the pages to the DHT based on a page's popularity (we decided later to insert all pages in our experiments), we added a field to the messages tables that indicates the number of views and modified the 'viewthread.php' script to increase that number for every page view. Also in the database, we created a new table called DHT_FILES that indicate which pages are inserted into the DHT (original message ID and page number) and the keys of these pages. We also record the number of posts in the page to indicate the version of the page currently stored in the DHT.

Another needed modification is to allow the ID of the original message, which is an auto-increment database field, to be specified by a POST parameter when creating a thread. This is needed in testing because subsequent replies need to know the ID of the original message in advance.

We also modified the 'reply.php' and 'newthread.php' scripts so that they redirect the caller to viewing the page of the new post directly after creating that post.
4.2.3 Inserting into the DHT

In our experiments we decided to setup the web application so it inserts all modified pages into the DHT. Since our 'dynamic' experiments begin by creating a thread and then a series of view and reply requests (as will be seen in the next chapter), inserting each modification into the DHT and then redirecting the user to the DHT would simulate the 'cache on access' approach. Remember that we modified our application to redirect the user that added a post to the modified page; hence the user would be the first to read the page from the DHT.

When form values are submitted to the 'newthread.php' or the 'reply.php' scripts, the site first updates the database with the new post. The site then retrieves the content of the new page using a PHP function that takes the URL of the 'viewthread.php' script. A parameter called NODHT is added to the URL which instructs the 'viewthread.php' script to return the actual file, not a redirection to the DHT. Then the site creates a TCP connection to a random DHT node using the DHashWeb port and sends a store command followed by the length of the file then the file contents to the node.

The site would then receive the key of the inserted file from the DHT peer, save the key to the database, and then redirect the client to the page (viewthread.php).

4.2.4 Redirecting to the DHT

As described previously in the approach chapter, when a user asks to view a page the user will be redirected to the DHT if the page is stored there. The web application can figure that out by checking the DHT_FILES table in the database using the message ID and the page number.
While our approach says that the redirection file will use a JavaScript for redirection, for simplicity in our implementation we will use an IFrame tag instead. This way the file stored in the DHT can be the actual html file returned to the user. The source of the tag would point directly to a random peer in the DHT so we won't have to actually implement the client component for our experiments.

While the IFrame approach might be acceptable to do our experiments, there are two main disadvantages that would prohibit us from adopting this approach in a real-world solution. The first one is that links inside the loaded IFrame would point to the address of the peer instead of the address of the site. The solution would be to use absolute links instead of relative ones.

The second disadvantage is that using JavaScript would mean that we can customize the stored page's content by using values of passed variables. While it's true that our simple application there is neither session management nor customizations per user, there is an important usage for JavaScript customizations. In this application, whenever a new page is added to a thread, all other pages would have to be updated to show the current number of pages in the thread. A link to the next page would also have to be added for the previously last page. We can achieve that without invalidating the old pages through a JavaScript that takes the number of pages in the thread and dynamically creates the navigation links.

Table 4.2 shows an example of a redirection HTML file returned by the 'viewthread.php' script to a random peer.

Table 4.2 – Example of a Redirection HTML File (IFRAME)

```
<html>
<body>
<iframe src="http://192.168.10.3:20003/gethtml?id=12345678901234567890
width = 100% height=100% frameborder=0">
```
4.3 The Testing Tool

To test our implemented system components, the web server and the DHT peers, we need a program that impersonates clients by generating HTTP requests. The program would also measure the latency that would be experienced by clients.

The tool we will use for this purpose is called httperf [37]. It is a tool used for measuring web server performance. Httperf provides many options for the construction of HTTP workloads. The number of connections to the web server and the rate (connection per second) at which these requests are created are specified as command line parameters. The target URL can be specified as a command line parameter or a log file can be provided which contains a list of URLs. Inside this log file, each URL can be associated with any HTTP method (e.g. GET or POST), and additional data (POST data) can be specified for each URL.

Generating more advanced workloads is possible. Bursts can be defined in the log file as an ordered sequence of calls to multiple URLs. An example of a burst is reading an HTML page and then the images that this page references. Httperf can emulate a browser by retrieving the HTML page and then issuing synchronous calls (on the same connection if it is still open) requesting images just after the HTML page is retrieved.

Log files can also contain definitions of sessions. A session is defined as a series of bursts, separated by user think time. When using sessions, the tester tries to emulate how visitors browse a site. Visitors usually retrieve a page and then take some time to read it (think time) and then request another page.
After running a workload using httperf, the program outputs many useful statistics about the applied workload. Table 4.3 shows an example of the output returned after running a workload defined using a single URL and four connections. However, we will not be using any of these. The average connection time could be used if we are testing requests for a single URL, but in our approach a request can cause a sequence of URLs to be called. Later in sub-section 4.3.3 we will show how we will calculate the end user latency.

**Table 4.3 – HTTPERF Sample Output**

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>httpperf --client=0/1 --server=www.google.com --port=80 --uri=/ --send-buffer=4096 --recv-buffer=16384 --num-conns=4 --num-calls=1</td>
</tr>
</tbody>
</table>

Total: connections 4 requests 4 replies 4 test-duration 0.946 s

Connection rate: 4.2 conn/s (236.5 ms/conn, <=1 concurrent connections)

Connection time [ms]: min 135.5 avg 236.5 max 463.7 median 152.5 stddev 153.5

Connection time [ms]: connect 7.0

Connection length [replies/conn]: 1.000

Request rate: 4.2 req/s (236.5 ms/req)

Request size [B]: 67.0

Reply rate [replies/s]: min 0.0 avg 0.0 max 0.0 stddev 0.0 (0 samples)

Reply time [ms]: response 229.5 transfer 0.0

Reply size [B]: header 680.0 content 222.0 footer 0.0 (total 902.0)

Reply status: 1xx=0 2xx=0 3xx=4 4xx=0 5xx=0

CPU time [s]: user 0.60 system 0.34 (user 63.6% system 36.0% total 99.7%)

Net I/O: 4.0 KB/s (0.0*10^6 bps)

Errors: total 0 client-timo 0 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 0 addrrunavail 0 ftab-full 0 other 0

The reply size could be used to measure the server's upload bandwidth when testing the client-server approach. However, it cannot be used when testing our approach mainly because part of the server's upload traffic is headed to DHT peers. Another problem with the reply size calculated by httperf is that it doesn't include the
size of TCP/IP headers and footers. We need to include the size of these headers and footers since they are part of the total Internet bandwidth used by any machine on the Internet. We will show in the next chapter how to measure the server's upload bandwidth using Linux IPTABLES.

While httperf does provide advanced options for the construction of workloads, our system has requirements that are not provided by httperf because httperf focuses on testing a single web server. Hence, we will modify the httperf implementation to extend its functionalities according to our purposes.

4.3.1 Basic Modifications

The first modification we did to httperf is to bound outgoing calls from httperf to one or more specific local IPs. The reason is that - as we will explain in next chapter - we ran several nodes (components) of our system on the same machine, and virtual local IPs were used to define artificial delays and bandwidth limits between these nodes.

The second problem with httperf is that it allows only a single web server to be specified for testing. Although you can pass various URLs in a log file to httperf, all of them must be located on the same server. Our modified version of httperf needs the ability to issue requests against our web server and the DHT peers. We kept the command line parameter that specifies the default target web server and added the ability to specify other web servers in the log file.

4.3.2 Modifying the Workload Generator

Since web pages contain embedded files like images, java scripts, and iframes, httperf gives the option to define sessions that would contain bursts of URLs to html files and URLs of embedded files, requests to embedded files are issued as soon as the
reply to the first URL - the html file - is received. We need such mechanism because a request to view a thread could redirect the user to a DHT peer. Also, creating a new thread or replying to an existing one redirects the user to the viewing URL.

There are two problems in redirection in our system that disallows us from using the httperf implementation of sessions/bursts. The first one is that in bursts, a single TCP connection is used to issue calls to the URLs in the burst. Httperf uses this approach, and since our bursts can contain URLs in different servers (the web server and the DHT peers), this approach will not work for us. We need to create a connection for each request or at least a new connection whenever the target web server changes.

The second problem we faced is that we may not be able to predict the target URL (the URL a client is redirected to) in advance. This is clear in case of redirection to DHT, where the URL specifies a DHT key which depends on the version of the page currently stored in the DHT. Since our experiments run many reply and view requests simultaneously, we cannot predict which version of the page a view request should return. That means that we cannot know the full URL to which a client will be redirected to in advance.

Our solution to the problem is to not specify the DHT URL in the log file passed to httperf. Instead, the web server returns a line in the HTTP header that instructs httperf to dynamically add a request to a certain URL in the current burst (after the current request). Examining the HTTP header would be much faster than parsing the whole HTML page returned.
4.3.3 Modifying Stats Collector

In addition to modifying how workloads are read from log files and how they are applied, we need to calculate the total connection time for each burst in the log file. This connection time is defined as the time elapsed since the first request in the burst is applied until the response for the last request in the burst is received. If any request in the bursts fails then the whole burst fails and we do not continue issuing requests in it.

We modified httperf to display the number of bursts that have succeeded, and the average total connection time for those successful bursts.
Chapter 5

EXPERIMENTING

To verify how our system will aid an overloaded dynamic web site, we ran some experiments which we will show in this chapter. We begin by examining the outline of the experiments. Then we will describe in details the testing environment, followed by the descriptions of the experiments. Finally, we will show the results of the experiments.

5.1 Experimenting Strategy

To test a distributed system like ours, two options are available. The first one is to simulate the system. P2psim [51] is a simulator that can be used for various DHT implementations. To use p2psim, its user must write the DHT algorithm in C++ code according to a template provided by p2psim, and the written code is not reusable in real implementations. P2psim simulates only the lookup layer and discards the storage layer. MACEDON [52] allows its users to write DHT algorithms using a special grammar and produces a simulation and real implementation code. However, MACEDON cannot be used to test already available implementations.

The other option to test distributed systems is to use emulation. ModelNet [53] is an emulator that can be used to evaluate DHT implementations. ModelNet allows its users to emulate of thousands of nodes using a dedicated network cluster. However, we needed a less complex approach that enables us to run a relatively small P2P network on a single or a few LAN machines. This was accomplished through the use of Linux traffic control modules, which we will explain in the next section.
All of the components of our system were installed in a single machine. The components we implemented in the previous chapter were run on this machine. These components include a web server, a number of DHT nodes, and a tool that generates clients' requests and measure various performance metrics. The environment was controlled to simulate an Internet-like network by setting delays and bandwidth limits between various components. The setup of the environment is described later in section 5.2.

Our system will be evaluated according to two main criteria. The first one is the latency experienced by end users. The latency is defined as the time interval that begins when a request for a page is created (when the user enters a URL or clicks on a link) to the time the page is fully received by the client's machine.

The other criterion is the total upload bandwidth of the server used during the experiment (or the average bandwidth per request). All of our experiments were executed twice, the first time with a single web server (client-server) approach, and the second time using our approach. Extracted values, the latency and the upload bandwidth, are compared to their counterparts in the other experiment.

We will show how these values will move when the load on the server is varied. We will vary the load by issuing URL requests at different rates (requests per second), which is possible through the use of the httperf tool we described in the previous chapter.

After that, we will modify various parameters to show how they would affect our results. These parameters include the ratio between reply and view requests, the number of DHT peers, and the method in which delay between nodes is created.
5.2 Setup of the Testing Environment

We will run our system, which consists of a web server, some DHT peers, and clients' requests generator, on a single machine. This is mainly possible because the number of DHT peers would be small (less than a hundred) in most of the targeted applications. Because of that we will experiment our system only on a small number of DHT peers. Other research efforts [31, 32] that used a similar approach to test DHTs were able to run 20-80 DHT nodes on a single machine. The specifications of our testing machine are indicated in Table 5.1.

Table 5.1. Testing Machine's Specifications

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel Core 2 Duo CPU P8700</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>3 GB DDR2</td>
</tr>
<tr>
<td>Hard Drive</td>
<td>SATA II 5400 rpm</td>
</tr>
<tr>
<td>OS</td>
<td>opensuse 11.2</td>
</tr>
</tbody>
</table>

5.2.1 Simulation of the Overlay Network

To be able to create an Internet-like environment on our single machine, we made use of Linux Traffic Control modules. The most important properties that need to be set are the delay between various nodes, and the bandwidth limit (upload/download) of each node. We ignored other properties like the probability of lost or duplicate packets or packets arriving in incorrect order.

5.2.2 IP Aliasing

The first step needed to setup traffic control is to assign different IP to each node (seen as a process by the OS) in our system. This can be done using IP aliasing provided by Linux, were you can assign multiple IPs for a single network adapter (the adapter on a single machine environment would be the loopback adapter). Each of our
components would need to support being bound to a specific local IP. This is the case in our Apache web server and in Chord nodes. We modified the httperf program used to generate client requests to support binding to a local IP address.

The two lines in Table 5.2 are the commands we use to create an IP alias for a single node. We used a bash script to create IPs for each node in our system.

Table 5.2 – Creating IP Aliases

<table>
<thead>
<tr>
<th>Command 1</th>
<th>Command 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifconfig lo:1 192.168.10.1</td>
<td>route add -host 192.168.10.1 dev lo:1</td>
</tr>
</tbody>
</table>

5.2.3 Traffic Control

Traffic Control is possible in Linux through the use of various types of queues that store outgoing packets from a specific network adapter [43]. The type of the queue (queue discipline) and its setup properties dictates when and how packets are enqueued to the queues and dequeued from them to be sent via the network adapter.

There are two main types (categories) of queues disciplines, classless and classfull. Classless are simple queues that are only defined by a few properties. The default queue discipline (or qdisc) used in a network adapter is a classless one called pfifo_fast. It contains three bands each of different priority and all work in a FIFO mode.

Classful queue disciplines are more complex; in these qdiscs, a tree of children classes can be defined, and each of these classes can contain a further qdisc. This allows complex behaviors to be applied to these qdiscs.

The main qdisc type we are going to use is a classful qdisc called the hierarchical token bucket (HTB) qdisc [44]. HTBs are usually used when there is a
fixed amount of bandwidth, which needs to be divided for different purposes, giving each purpose a guaranteed bandwidth. To define each purpose we use a class that inherits from the base HTB qdisc. Filters are used to define to which class packets should go. Filters can match properties such as the source or destination IP or port.

The code in Table 5.3 illustrates how we can use HTBs to specify upload bandwidth limits and packets delay for a node in our system.

Table 5.3 – HTB Queuing Disciplines

<table>
<thead>
<tr>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc qdisc add dev lo root handle fffe: htb</td>
</tr>
<tr>
<td>tc class add dev lo parent fffe: classid fffe:1 htb rate 1mbit</td>
</tr>
<tr>
<td>tc qdisc add dev lo parent fffe:1 handle 1: netem delay 100ms</td>
</tr>
<tr>
<td>tc filter add dev lo parent fffe: protocol ip prio 1 u32 match ip src 192.168.10.1 flowid fffe:1</td>
</tr>
</tbody>
</table>

5.2.4 Setting Upload Bandwidth and Delays

All research projects attempting to simulate or emulate a peer to peer network need to set hypothetical delays between the peers. Many projects use what is known as the King matrix [34], which contains measurements of network latencies between arbitrary Internet end hosts. We use this matrix to define delays between nodes in our system. These nodes include the web server and the DHT peers, and a group of nodes that represents the clients. Since the clients cannot be assumed to be located in a single location, we will modify the httperf tool to send requests from different IPs (10 IPs).

However, in contrast to other projects which model a peer to peer network, we have a centralized component which is the web server. This means that the results could depend heavily on the location of the server in the matrix. Because of this, we repeat some of the experiments using a static latency between nodes (RTT of 200ms).
These experiments should give results that depend more on the number of hops messages traverse than on the actual latencies between these hops.

The upload speed of the web server is set to 1mbps, while the DHT peers were limited to 512kbps. Linux traffic control does not support shaping of incoming traffic. It supports only dropping incoming packets, not delaying them. Because of that, we neglect the effect of the download bandwidth limits of nodes. This assumption should not affect our results, especially if we assumed that DHT peers use ADSL Internet connections, in which the download speeds are typically four times higher than the upload speeds.

We used a bash script to assign these delays and bandwidth limits. The example we have previously shown in Table 5.3 illustrates how we can delay and shape (limit the rate of) the packets going out of a specific IP. These commands can be used when we experiment while using fixed delays between nodes in the system. However, when the King's matrix is used, the delay of the packets originating from an IP depends on the destination. Because of that, we have to create a qdisc between every two nodes in the system.

To do that, we created a program that reads the King's matrix and creates a bash script file that limits bandwidth and adds delays between each two nodes in the system. Table 5.4 shows a subset of the long script created using our program. The subset includes the commands needed to limit packets from a single node going to two other nodes.

The first line creates the main HTB qdisc. For each node (IP) in our system, we create a class that inherits from the main qdisc (shown in the second line). This line specifies the total upload bandwidth for packets leaving the node IP. Next, for each
possible target IP, three lines are added. The first line creates a child class which limits the rate to a small value. Children of a parent class can borrow bandwidth of each other as long as their total bandwidth does not exceed the bandwidth limit of their parent. The next line attaches a netem qdisc to the class that adds the delay. The last line declares a filter which sends packets - originating from the node IP to the target IP – to the netem qdisc just declared.

**Table 5.4 – Limiting Traffic When Using the King's Matrix**

<table>
<thead>
<tr>
<th>tc qdisc add dev lo root handle ff: htb</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc class add dev lo parent ff: classid ff:5 htb rate 512kbit</td>
</tr>
<tr>
<td>tc class add dev lo parent ff:5 classid ff:502 htb rate 5kbit ceil 512kbit</td>
</tr>
<tr>
<td>tc qdisc add dev lo parent ff:502 handle 502: netem delay 61ms</td>
</tr>
<tr>
<td>tc filter add dev lo parent ff: protocol ip prio 1 u32 match ip src 192.168.10.5 match ip dst 192.168.10.2 flowid ff:502</td>
</tr>
<tr>
<td>tc class add dev lo parent ff:5 classid ff:503 htb rate 5kbit ceil 512kbit</td>
</tr>
<tr>
<td>tc qdisc add dev lo parent ff:503 handle 503: netem delay 59ms</td>
</tr>
<tr>
<td>tc filter add dev lo parent ff: protocol ip prio 1 u32 match ip src 192.168.10.5 match ip dst 192.168.10.3 flowid ff:503</td>
</tr>
</tbody>
</table>

**5.2.5 Measuring the Web Server's Upload Bandwidth**

Since a part of the web server upload bandwidth is used to communicate with the DHT, we cannot measure the total upload bandwidth of the web server from the client side. To accomplish this task, we rely on iptables, which is a program that allows a system administrator to configure the Linux IPv4 packet filtering ruleset [38] (configure Linux firewall). Using iptables, we create a new rule that matches packet originating from the web server IP address. The rule does not specify a target, since we are only interesting in counting the bytes leaving that IP. Table 5.5 shows the commands we used to create the rule and to display the number of bytes matched by that rule.
5.3 Experiments

We needed to apply various workloads on our implemented system to test its performance. First, we retrieved a dataset which contain a number of threads, each contain a number of text posts. The dataset was downloaded from [35].

We used the dataset to construct our workloads. Each workload consists of a log file that contains a series of URLs in separated lines. These URLs will be issued to the web server at a specific rate, and we will measure the time to reply when our DHT peers are used and when they are not. We will first test on static content, and in this case all of these URLs will be view requests. Then, we will test on dynamic content, and in this case these URLs would be either view requests or reply requests.

In case of view requests, the URL specifies the ID of the thread in addition to the page number. In case of reply requests, the line contains post data such as the title, the text of the post, the author, and the ID of the thread. Since each reply request results in a redirection to the view page, we always insert a view request after every reply request. The line that contains the mentioned view request begins by a space, to indicate that the view request and the reply one belong to the same burst.

We need bursts to be defined so that the latency is calculated as the summation of the latency of all requests in the burst. When DHT is enabled (testing our approach), all requests – reply and view – contain a get from DHT request in the same burst. In other words, we generate bursts of exactly two (view) or three (reply) requests. The actual log file doesn't contain the DHT requests; they are inserted by the

<table>
<thead>
<tr>
<th>Table 5.5 – Creating and Examining an IPTABLES rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>#creating the rule</td>
</tr>
<tr>
<td>iptables –I OUTPUT –s 192.168.10.2</td>
</tr>
<tr>
<td>#examining the rule</td>
</tr>
<tr>
<td>iptables –L OUTPUT –v –n -x</td>
</tr>
</tbody>
</table>
httpperf tool when the actual redirection happens. The reason, as we explained in the implementation chapter, is that we cannot know the key of the requested page at time of the creation of the log file. Table 5.6 shows an example of a view request followed by an example of a reply burst.

These log files are created using a helper program we created that reads an XML file of our dataset, and creates a log file that contain view and reply requests according to some given parameters. These parameters include the total number of requests and the percentage of reply requests; the order of reply and view requests is randomized.

**Table 5.6 – HTTPERF Log File**

<table>
<thead>
<tr>
<th>Request 1</th>
<th>Request 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>/exp/viewthread.php?mid=100</td>
<td>/exp/reply.php method=POST</td>
</tr>
<tr>
<td></td>
<td>contents='FormName=reply&amp;FormAction=insert&amp;topic=RE%3a+Title+of+the+post&amp;author=Ryan&amp;message=This+is+the+text+of+the+message&amp;Rqd_mid=1000'</td>
</tr>
<tr>
<td>/exp/viewthread.php?mid=100</td>
<td></td>
</tr>
</tbody>
</table>

Before running any experiment, we prepare our testbed by creating the IP aliases and the queues disciplines that limit traffic between the IPs using two bash scripts. In case we are testing our system, we run the DHT daemons for every DHT peer. After that, we run our DHashWeb program for every peer. Both these steps also use bash scripts that loops on the number of peers. Table 5.7 shows the command we use to run a single DHT peer. We show the command for running the web server's DHT node and the command for running a normal DHT peer.

**Table 5.7 – Running a DHT peer**

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Web Server's DHT node</td>
</tr>
<tr>
<td>lsd/start-dhash --root dhash-2 --s 2 --j 192.168.10.2:10000 --l 192.168.10.2 --p 1000</td>
</tr>
<tr>
<td># DHT Peer</td>
</tr>
<tr>
<td>lsd/start-dhash --root dhash-3 --s 2 --j 192.168.10.2:10000 --l 192.168.10.3</td>
</tr>
</tbody>
</table>
The first node is the gateway to the DHT hence it binds itself to a specific port using the "--p" parameter. The line also specifies the local IP which the node is bound to and the directory used to save the node's database and sockets. These two parameters (-l and --root) are different in all nodes. The "-s" parameter specifies the selected routing protocol, which is Accordion in our case.

When running DHT based experiments, we restart all the DHT peers after every experiment, to ensure that all cached data are deleted.

In our experiments, we insert the first message manually before applying the workload. When running the workload via httpperf, several parameters are specified as command line parameters. Table 5.8 shows a sample httpperf command we used to run an experiment. The server's parameter specifies the IP address of the web server. The rate parameter specifies the rate at which bursts are sent. The rate is specified as a number of bursts per second, which httpperf converts it to interval between bursts. The wsesslog parameter specifies the name of the log file in addition to the number of bursts to run. Other parameters shown in the command line are repeated for all experiments. The output of httpperf is saved into a results file which will be processed later.

<table>
<thead>
<tr>
<th>Table 5.8 – Running an Experiment using HTTPERF</th>
</tr>
</thead>
<tbody>
<tr>
<td>httpperf --server 192.168.10.2 --rate=8 --wsesslog=480,1,106531.log --hog --add-header &quot;Content-type: application/x-www-form-urlencoded\n&quot; &gt; res_dn_8</td>
</tr>
</tbody>
</table>

We choose the number of bursts so that the experiment with the highest rate lasts for 60 seconds. We chose 60 seconds because increasing the length of the experiment to higher values could cause many requests to time out under high rates.
In the next section, we will show the results of our experiments that targeted both static content (no reply requests) and dynamic content. We will test the effect of using the same delay between all hops instead of King's matrix. Also we will test the effect of changing the number of DHT peers (first experiments use 10 DHT peers).

5.4 Results and Analysis

The first set of experiments tries to test how our approach would function in case of static content. In these experiments, we apply only view requests on a thread page. In the second set, both view and reply requests are included in our workloads.

5.4.1 Experimenting on Static Content

In the first experiment, we tested using ten DHT nodes. The size of the static file is about 25KB. The number of view requests is 360, and the shortest experiment (highest rate) lasts for 60 seconds.

Figure 5.1 shows the results when testing using the King's matrix (for latencies between nodes). Results show that the normal client-server approach performs slightly better under low load (less than 4 requests/seconds). As the rate increases further, the latency of our approach is almost steady while the latency of the client-server approach increases quickly. The latency of our approach would increase by higher margins if the upload bandwidth of the web server or the total upload bandwidth of the peers becomes a bottleneck. This is possible if the rate of requests increases further or if the number of peers decreases. Later on (Sub-section 5.4.3) we will show the results obtained when we vary the number of DHT peers while fixing the requests rate.
Figure 5.1 – Latency in Static Experiments, 25KB, using King’s Matrix

Figure 5.2 shows the results we obtained when we used a fixed delay between all nodes in the system instead of the King's matrix. Under low rates, the client-server approach is twice as fast when compared to our approach. Under higher rates, values of latencies are close to those of the previous experiment.

Figure 5.2 - Latency in Static Experiments, 25KB, using Fixed Delays
The reason of the difference between the two cases above lies in the location of the server in the King's matrix. We selected the first node (row and column) of the matrix to represent the server; and the delay between this node and some of the nodes that represent clients is high. This delay contributes to a large proportion of the total latency in the first figure; and this is the cause that both approaches result in close results under low load.

In the next experiment we increased the size of the document to about 50KB (doubled the size). We ran the same number of view requests and we used fixed delay between the nodes. Figure 5.3 shows the latencies we calculated.

![Figure 5.3 - Latency in Static Experiments, 50KB file, using Fixed Delays](image)

The results show that the client-server system can support lower rates than previous experiments, since the upload bandwidth of the server is the bottleneck. Under low load, the latencies are close to the values we retrieved in the previous experiment for both the client-server approach and ours.
Table 5.9 shows the average upload bandwidth consumed by the server for all the three experiments, calculated per request, and the saving in the server's upload bandwidth when using our approach instead of the client-server one. We used the results obtained when applying the highest request rate. Changing the rate has negligible effect on the server's total upload bandwidth.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Client - Server</th>
<th>Our Approach</th>
<th>Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>25KB, King's Matrix</td>
<td>28,025</td>
<td>844</td>
<td>%96.99</td>
</tr>
<tr>
<td>25KB, Fixed Delay</td>
<td>28,062</td>
<td>844</td>
<td>%96.99</td>
</tr>
<tr>
<td>50KB, Fixed Delay</td>
<td>41,622</td>
<td>844</td>
<td>%97.97</td>
</tr>
</tbody>
</table>

The results in the table show that, as the average size of the web files sent to clients increase, our approach would enable the server to support a larger number of clients (under the same request rate).

5.4.2 Experimenting on Dynamic Content

In this set of experiments we apply requests that modify data and requests that read the modified data. We start with threads with a single post (before running the experiment), and apply a group of reply and view requests from a log file to the site. As in the experiments we done on static content, the rate of requests is modified per experiment. The ratio between view and reply requests is fixed, and the order of these requests is randomized. The number of DHT peers is ten.

In the first experiment we applied 480 requests, using the King's matrix to specify delays between nodes. The size of the complete thread page is about 25KB.
The ratio of reply requests is 5%. Figure 5.4 shows the measured latencies. Figure 5.5 shows the results of the same experiment repeated using fixed delays.

Both experiments results in about 91.5% saving of the web server's upload bandwidth when using our approach. The rates supported in dynamic experiments are
higher than static ones. The primary reason is that we start with threads with a single post. Hence, the average size of the files returned in dynamic experiments is much lower than the sizes in static experiments.

### 5.4.3 Varying the Number of Peers

In the next experiments, we will fix the rate of requests, and change the number of DHT peers. We will experiment on a number of peers that range from two (which is the minimum since we have three replicas per file, and the server is part of the DHT network) to thirty.

The first experiment was applied on static content. We used a single file with size around 25KB, and applied requests at a rate of six per second. We used fixed delays between nodes in our system. Figure 5.6 shows the results. The client-server approach does not depend on the number of peers hence the latency is shown as a straight line in the figure.

![Figure 5.6 – Latency in a Variable Number of Peers, Static Content](image-url)
The second experiment was applied on dynamic content, with a fixed request rate of eight per second. Fixed network delays were used between nodes. Results are shown in Figure 5.7.

![Figure 5.7 – Latency in a Variable Number of Peers, Dynamic Content](image)

The two experiments show different behaviors at both low number and high number of peers. When experimenting on two DHT peers, the total upload bandwidth of the two peers is equivalent to the upload bandwidth of the server. Hence, since the server redirects all requests to the peers, we expect the performance of the two systems (client-server and our system) to be close.

This does not mean that our system cannot take advantage of a small number of peers. However, in a real world implementation, we expect the server to handle a subset of requests when the load is average or all of them if the load is low. In our experimenting implementation the server redirects all the incoming requests which mean that a large portion of the server's upload bandwidth is not utilized.
In the static experiment, the latency using two peers is slightly higher than the latency in the client-server approach. The difference mainly comes from the delay wasted until the client is redirected to peers and from the imperfect distribution of load between the two peers. In the dynamic experiment, the latency in the client-server is clearly lower than that of our approach when using two peers. In addition to the reasons we stated for the static experiment, the main reason for the difference comes from reply requests.

In our approach, reply requests are delayed until the server sends the new file to a DHT node. After that the client who sent the reply is redirected to the newly inserted file. This cause the reply requests to take considerable longer than they would in the client-server approach. Note that this approach can be modified by returning the new file directly from the server to the client that issued the reply request. This would speed up the reply requests, at the expense of increasing the used upload bandwidth of the server.

When we increase the number of peers, our dynamic experiment shows that the latency always decreases. However, in the static experiment, the latency when using thirty peers is slightly higher than when using twenty peers and close to the latency obtained when using only ten nodes.

Before trying to explain the reason behind the different behavior in the two experiments, let's try to analyze what increasing the number of peers would lead to. First, the obvious benefit is to increase the total bandwidth of the system. This would help if the load is higher than the sum of the capabilities of the DHT peers. Also, the files in the DHT would be distributed between more peers hence decreasing the average load on a single peer. A single file is still however kept by three nodes that
hold the replicas, which means that increasing the number of peers won't help peers responsible for a single popular file.

In fact, for a single file, increasing the number of peers would reduce the probability of a cache hit when a client gets redirected to a random peer. The second possible disadvantage in increasing the number of peers would be the increase of query latency. Many DHTs resolve queries in O(log N), where N is the number of peers in the DHT. However, we use Accordion in our experiments, which would resolve queries in O(1) for small number of peers. Hence, we expect most queries to be resolved in one or two hops.

The first disadvantage is probably the reason behind the decrease performance at 30 peers in the static experiment. Since we only experiment on a single thread, the percentage of a cache hit would decrease. This would result in higher latency since the file must be obtained from one of the three replicas before being forwarded to the client. This also means that the load on the peers holding the three replicas would increase.

While the dynamic experiments were applied also on a single thread, the thread file is repeatedly modified. Whenever the file is modified, the corresponding DHT key is modified too, and the old cached copies in the DHT become invalid. Because of this, the load on replica peers decreases, and the benefit of caching decreases. This would be the major reason behind the difference in results between the static and dynamic experiments.
Chapter 6

CONCLUSION AND FUTURE DIRECTIONS

This dissertation proposed a new approach for cooperative web caching, where a web site can make use of a group of volunteer PCs to increase its serving capabilities. We proposed some new ideas in this work; some of these ideas try to enable customizations to pages that were not possible using many other similar projects. Also we tried to provide better support for dynamic sites where content can change frequently.

We tested some of these ideas, and showed how using our approach would reduce the latency and the probability of errors experienced by visitors of the site when the site is under high load. Even under low load, the increase in latency is acceptable. Thus, we would choose to enable the DHT whenever the server senses that the load begins to increase.

We showed also that the upload bandwidth used by the server is decreased by a large percentage when our approach is used. This enables the server to support a much larger number of visitors than in a single web server approach. The actual number depends on many factors. These factors include the size of invalidated pages per time unit, the upload speed of the server, and the total upload speed of the peers. Thus, the new bottleneck in the system could be either the upload speed of the server or the total upload speed of the peers (or maybe the processing power of the server).

We illustrated and tested an Internet forums application. Our approach could be applied to various types of dynamic applications, but probably not all. Other type of
applications would need investigation to find out if our approach performs satisfactory or needs some modifications.

Internet forums and similar applications are primary candidates of using some ideas illustrated in our approach like incremental updates. This feature is possible because in forums pages are built incrementally, and this feature could be used to reduce the bandwidth requirements of the server. However, careful analysis is needed to determine when to use it (is it better to upload the whole page to the DHT), and how to use it (should we put the incrementing JavaScript in DHT or not).

Also, in forums, any update would cause the updated (invalidated) page to be returned to the user. For example when a user adds a reply the site would return the new page with the added reply. Because of this our push approach could work like a "cache on access" approach, and this ensures that any page uploaded to the DHT will not be a waste. The only effect is that reply requests would be delayed a little until the page is written to the DHT.

In case these properties do not exist in the targeted application, modifications to our approach could get better results. One suggestion is the usage of a pull approach while storing links to cached copies in the DHT. Another scenario when this could be needed is the case where the rate of updates is high and the number of volunteers is low. In this case, writing a number of replicas to the DHT would waste valuable bandwidth resources.

Another issue that would need more investigating in our push approach is what pages to upload the DHT. We would want to find out if there are conditions that would dictate than we don’t push an updated page to the DHT. And if a page is inserted in the DHT, on what basis should we choose the expiry date of the DHT
block. Using high values would increase the requirements of DHT peers in terms of disk space and bandwidth. Also, other alternatives to using content hashing as key of DHT blocks could be investigated. The primary reason would be to reduce these requirements.
APPENDIX A

Source Code of DHT Program Modifications

Table A.1 – DHashWeb.C

```c
#include <async.h>
#include <dhash_common.h>
#include <dhashclient.h>
#include <dhblock.h>
#include <dhblock_keyhash.h>
#include <sfscrypt.h>
#include <sys/time.h>
#include <libadb.h>

#define BUFSIZE 8096

int webport = -1;

u_int64_t getusec ()
{
    timeval tv;
    gettimeofday (&tv, NULL);
    return tv.tv_sec * INT64(1000000) + tv.tv_usec;
}

class DHashWeb
{
    public:
    str ctlsock;
    str dbsock;
    dhashclient *dhash;
    adb *db;
    adb *cache_db;

    DHashWeb(str csk,str dbsk)
    {
        ctlsock = csk;
        dbsock = dbsk;
        dhash = NULL;
    }

    ~DHashWeb() { if(dhash) { delete dhash; dhash = NULL; } } 

    void echo_write(int fd, strbuf buf);
    void echo_read(int fd, ref<int> cont);
    void web_accept_connection(int fd);
    void startweb ();
    void Connect();
    void eofhandler ();
    void fetchcb (u_int64_t start, int fd, chordID id, int retry, dhash_stat stat, ptr<dhash_block> blk, vec<chordID> path);
    void storecb (int fd, strbuf id, u_int64_t start,dhash_stat status,
```
ptr<insert_info> i);

    void ConnectLocalDB();
    void local_fetch_cb(int fd, strbuf buf, chordID id, adb_status stat,
                        adb_fetchdata_t id);
    void cache_fetch_cb(int fd, strbuf buf, chordID id, adb_status stat,
                        adb_fetchdata_t id);
    void cache_store_cb(adb_status error);

    void retry_retrieve(int fd, chordID id, int retry);
    void buffered_fill(strbuf *pbuf, str *s);
};

void DHashWeb::echo_write(int fd, strbuf buf)
{
    int n = buf.tosuio()->output(fd);
    if(n < 0)
        fatal << "error in write\n";
    if(buf.tosuio()->resid())
        return;

    fdcb(fd, selwrite, 0);
    fdcb(fd, selread, 0);
    close(fd);
}

void DHashWeb::buffered_fill(strbuf *pbuf, str *s)
{
    char *p = (char*) s->cstr();
    char mybuf[BUFSIZE+1];
    int len = s->len();

    while(len>0)
    {
        int csize = min(len,_BUFSIZE);
        memcpy(mybuf,p,csize);
        mybuf[csize] = 0;
        len -= csize;
        p+= csize;
        str x(mybuf,csize);
        (*pbuf) << x;
    }
}

void DHashWeb::echo_read(int fd, ref<int> rem)
{
    int i;
    char buffer[BUFSIZE+1];
    char *fstr;
    char id[100];

    str tmp;
    strbuf buf;
strbuf buf2;

static strbuf st_buf; // buffer not yet processed from last reading operation

int n = buf.tosuio()->input(fd);

if(n < 0)
    fatal << "read\n";

if(n == 0) {
    fdcb(fd, selread, 0);
    close(fd);
    return;
}

tmp = buf;
strcpy(buffer,tmp.cstr());

if(strncmp(&buffer[0], "STORE", 5) == 0 || *rem > 0 )
{
    fstr = (char*) tmp.cstr();
    if (*rem <= 0) { // encountered STORE
        fstr += 6; // bypass the STORE
        *(fstr+7)=0;
        int filelen = atoi(fstr); //read length of the file
        fstr+=8;
        n -= 14;
        *rem = filelen;
    }

    st_buf << fstr;

    if ( n >= *rem )
    {
        str x = st_buf;
        chordID id = compute_hash(x.cstr(),strlen(x.cstr()));
        //buf2 << "OK " << id;
        buf2 << id;

        fdcb(fd, selwrite, wrap(this,&DHashWeb::echo_write, fd, buf2));
        dhash->insert(x.cstr(),strlen(x.cstr())),wrap(this,&DHashWeb::storecb,fd,buf2,getusec());
        *rem = 0;
        st_buf.tosuio()->clear();
    }
    else
    {
        *rem = *rem - n; // wait until we get the remaining of the file*/
        return;
    }
}

// Don't read any more data on this socket
fdcb(fd, selread, 0);
for (i=0; i<n; i++) // remove CF and LF characters
    if (buffer[i] == '\r' || buffer[i] == '\n')
        buffer[i] = '*';

if (strncmp(buffer, "GET ", 4) &&
    strncmp(buffer, "get ", 4)) {
    buf2 << "HTTP/1.0 404 Not Found\r\nContent-Type: " << "text/html" <<
    "\r\n\r\n";
    buf2 << "<HTML> <BODY> SORRY, only get operations are supported</BODY> </HTML>";
    fdcb(fd, selwrite, wrap(this, &DHashWeb::echo_write, fd, buf2));
    return;
}

for (i=4; i<BUFSIZE; i++) {
    // null terminate after the second space to ignore extra stuff
    if (buffer[i] == ' ') {
        // string is "GET URL " + lots of other stuff
        buffer[i] = 0;
        break;
    }
}

if (strncmp(&buffer[0], "GET /gethtml?id=", 16)) {
    buf2 << "HTTP/1.0 404 Not Found\r\nContent-Type: " << "text/html" <<
    "\r\n\r\n";
    buf2 << "<HTML> <BODY> SORRY, operation not supported</BODY> </HTML>";
    fdcb(fd, selwrite, wrap(this, &DHashWeb::echo_write, fd, buf2));
    return;
}

fstr = &buffer[0];
fstr += 16;
strcpy(id, fstr);
chordID cid = bigint(id, 16);

// Try to fetch from the cache
cache_db->fetch(cid, wrap(this, &DHashWeb::cache_fetch_cb, fd, buf2, cid));

void DHashWeb::web_accept_connection(int fd) {
    struct sockaddr_in sin;
    unsigned sinlen = sizeof(sin);

    int cs = accept(fd, (struct sockaddr *) &sin, &sinlen);
    if (cs >= 0) {
        // warn << "accepted connection. file descriptor = " << cs << "\n";
    } else if (errno != EAGAIN)
        fatal << "accept; errno = " << errno << "\n";
    ref<int> rem = New refcounted<int>;
```cpp
*rem = 0;
fdcb(cs, selread, wrap(this,&DHashWeb::echo_read, cs, rem));
}

void DHashWeb::startweb ()
{
    int fd = inetsocket(SOCK_STREAM, webport);
    if (fd < 0)
        fatal << "inetsocket\n";
    make_async(fd);
    if (listen(fd, 50) < 0)
        fatal << "listen\n";
    fdcb(fd, selread, wrap(this,&DHashWeb::web_accept_connection, fd));
}

void DHashWeb::Connect()
{
    dhash = New dhashclient (ctlsock);
    dhash->seteofcb (wrap (this, &DHashWeb::eofhandler));
    delaycb(0,wrap(this, &DHashWeb::startweb));
}

void DHashWeb::eofhandler ()
{
    warn << "Exiting\n";
    delete this;
    exit (0);
}

void DHashWeb::storecb (int fd, strbuf id, u_int64_t start,dhash_stat status, ptr<insert_info> i)
{
}

void DHashWeb::retry_retrieve(int fd, chordID id, int retry)
{
    dhash->retrieved(id,DHASH_CONTENTHASH,wrap(this,&DHashWeb::fetchcb,getusec(),fd, id, retry));
}

void DHashWeb::fetchcb (u_int64_t start, int fd, chordID id, int retry, dhash_stat status, ptr<dhash_block> blk, vec<chordID> path)
{
    strbuf buf2;
    if (stat || !blk)
    { // failed to fetch the block
        if (retry > 2)
        {
            buf2 << "HTTP/1.0 404 Not Found\r\n\r\n" << "Content-Type: " << "text/html"
```
buf2 << "<HTML> <BODY> SORRY, file not found</BODY> </HTML>";
    fdcb(fd, selwrite, wrap(this,&DHashWeb::echo_write, fd, buf2));
}  
else //retry  
    delaycb(1,0,wrap(this,&DHashWeb::retry_retrieve,fd,id,retry+1));
} else  
{
    buf2 << "HTTP/1.0 200 OK
Content-Type: " "text/html" ?
    buffered_fill(buf2,blk->data);
    fdcb(fd, selwrite, wrap(this,&DHashWeb::echo_write, fd, buf2));
    cache_db->store(id,blk->data,wrap(this,&DHashWeb::cache_store_cb));
}

void DHashWeb::cache_fetch_cb(int fd, strbuf buf2, chordID cid, adb_status stat, adb_fetchdata_t t)
{
    if (stat == ADB_OK) // cache hit  
    {
        buf2 << "HTTP/1.0 200 OK
Content-Type: " "text/html" ?
        buffered_fill(buf2,t.data);
        fdcb(fd, selwrite, wrap(this,&DHashWeb::echo_write, fd, buf2));
    } else // cache miss  
    {
        // Check the local DB
        db->fetch(cid,wrap(this,&DHashWeb::local_fetch_cb,fd,buf2,cid));
    }
}

void DHashWeb::local_fetch_cb(int fd, strbuf buf2, chordID id, adb_status stat, adb_fetchdata_t t)
{
    if (stat == ADB_OK)  
    {
        buf2 << "HTTP/1.0 200 OK
Content-Type: " "text/html" ?
        buffered_fill(buf2,t.data);
        fdcb(fd, selwrite, wrap(this,&DHashWeb::echo_write, fd, buf2));
    } else  
    {
        dhash->retrieve(t.id,DHASH_CONTENTHASH,wrap(this,&DHashWeb::fetchcb,getusec()),fd, id , 0);
    }
}

void DHashWeb::cache_store_cb(adb_status error)
void DHashWeb::ConnectLocalDB()
{
    db = New adb (dbsock,"chash.c",false,NULL);
    cache_db = New adb (dbsock,"ccache",false,NULL);
}

void cleanup (DHashWeb *dh)
{
    if (dh)
    {
        delete dh;
        exit (0);
    }
}

int main (int argc, char **argv)
{
    if (argc < 4)
    {
        fatal << "you need to provide the dhash socket, web port, and adbd
socket\n";
        return 1;
    }

    webport = atoi(argv[2]);
    setprograme (argv[0]);

    DHashWeb *dh = New DHashWeb(argv[1],argv[3]);

    sigcb (SIGTERM, wrap (&cleanup, dh));
    sigcb (SIGINT, wrap (&cleanup, dh));
    sigcb (SIGHUP, wrap (&cleanup, dh));

    dh->Connect();
    dh->ConnectLocalDB();

    amain ();
}
APPENDIX B

Source Code of Testing Tool Modifications

Table B.1 – wsesslog.h

```c
#ifndef wsesslog_h
#define wsesslog_h

typedef struct req REQ;
struct req
{
    REQ *next;
    int method;
    char *uri;
    int uri_len;
    char *contents;
    int contents_len;
    char extra_hdrs[50]; /* plenty for "Content-length: 1234567890" */
    int extra_hdrs_len;
    char server[20];
    int port;
    Time birth_time;
    Time lifetime;
};

typedef struct burst BURST;
struct burst
{
    BURST *next;
    int num_reqs;
    Time user_think_time;
    REQ *req_list;

    int num_calls;
    REQ *cur_req;
    int failed;
    Time birth_time;
    Time lifetime;
};
#endif
```

Table B.2 – wsesslog.c

```c
#include <assert.h>
#include <ctype.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <httperf.h>
```
#include <conn.h>
#include <core.h>
#include <event.h>
#include <rate.h>
#include <session.h>
#include <timer.h>
#include <wsesslog.h>

#ifndef TRUE
#define TRUE (1)
#endif

#ifndef FALSE
#define FALSE (0)
#endif

BURST *first_burst, *cur_burst;

#define CONN_PRIVATE_DATA(c) \((Conn_Private_Data *) ((char *)(c) + conn_private_data_offset))

#define CALL_PRIVATE_DATA(c) \((Call_Private_Data *) ((char *)(c) + call_private_data_offset))

typedef struct Conn_Private_Data
{
    Call *call;
    /*Sess *sess; struct Conn_Info *ci;*/
} Conn_Private_Data;

typedef struct Call_Private_Data
{
    BURST *burst;
    REQ *req;
} Call_Private_Data;

static size_t conn_private_data_offset = -1;
static size_t call_private_data_offset = -1;

size_t wsesslog_call_priv_offset; // we want to pass this to the stat collector

/* Methods allowed for a request: */
enum
{
    HM_DELETE, HM_GET, HM_HEAD, HM_OPTIONS, HM_POST, HM_PUT, HM_TRACE,
    HM_LEN
};

static const char *call_method_name[] =
{
    "DELETE", "GET", "HEAD", "OPTIONS", "POST", "PUT", "TRACE"
};

static int num_bursts_generated;
static int num_bursts_destroyed;
static Rate_Generator rg_sess;
static int total_num_bursts;

/* Allocates memory for a REQ and assigns values to data members.  
   This is used during configuration file parsing only.  */
static REQ*
new_request (char *uristr)
{
    REQ *retptr;

    retptr = (REQ *) malloc (sizeof (*retptr));
    if (retptr == NULL || uristr == NULL)
        panic ("%s: ran out of memory while parsing %s\n", 
               prog_name, param.wsesslog.file);

    memset (retptr, 0, sizeof (*retptr));
    retptr->uri = uristr;
    retptr->uri_len = strlen (uristr);
    retptr->method = HM_GET;
    retptr->server[0] = 0;
    retptr->port = -1;
    return retptr;
}

/* Like new_request except this is for burst descriptors.  */
static BURST*
new_burst (REQ *r)
{
    BURST *retptr;

    retptr = (BURST *) malloc (sizeof (*retptr));
    if (retptr == NULL)
        panic ("%s: ran out of memory while parsing %s\n", 
               prog_name, param.wsesslog.file);

    memset (retptr, 0, sizeof (*retptr));
    retptr->user_think_time = param.wsesslog.think_time;
    retptr->req_list = r;
    retptr->failed = 0;
    return retptr;
}

void burst_failure(BURST *b)
{
    Any_Type arg;
    arg.l = 0;
    b->failed = 1;
    num_bursts_destroyed++;
    event_signal(EV_SESS_FAILED,(Object *) b, arg);

    if (num_bursts_destroyed >= param.wsesslog.num_sessions)
        core_exit();
}

static void send_call(Call *call, REQ *req, BURST *b)
{
Conn *con;
Conn_Private_Data *priv;

con = conn_new();
if (!con)
{
    burst_failure(b);
    return;
}

if(req->port != -1)
{
    con->hostname = strdup(req->server);
    con->hostname_len = strlen(req->server);
    con->port = req->port;
    core_addr_intern(con->hostname,con->hostname_len,con->port);
}

// we must set here private data, conn must point to call
priv = CONN_PRIVATE_DATA (con);
priv->call = call;

// connect first
if (core_connect (con) < 0)
    burst_failure(b); // this burst failed

// calls will be issued when we receive the CONN_CONNECTED event

static void issue_call(BURST *burst)
{
    REQ *req;
    Call *call;
    Call_Private_Data *priv;
    const char *method_str;

    if(burst->num_calls == 0) // first call in the burst
        burst->cur_req = burst->req_list;

    if( (req = burst->cur_req) != 0) // burst didn't finish
    {
        burst->cur_req = req -> next; // advance to next request for next time
        burst->num_calls ++;

        call = call_new ();
        if (!call)
        {
            burst_failure(burst);
            return;
        }

        method_str = call_method_name[req->method];
        call_set_method (call, method_str, strlen (method_str));
        call_set_uri (call, req->uri, req->uri_len);
        if (req->contents_len > 0)
        {
            // add "Content-length:" header and contents, if necessary:
            call_append_request_header (call, req->extra_hdrs,
req->extra_hdrs_len);
call_set_contents (call, req->contents, req->contents_len);
}

priv = CALL_PRIVATE_DATA(call);
priv->burst = burst;
priv->req = req;

if ( 0 ) //DBG > 0)
    fprintf (stderr, "%s: accessing URI `%s'
    " prog_name, req->uri);

//now issue the call via a new connection:

req->birth_time = timer_now();
send_call(call, req, burst);
}

static void
call_rcv_header(Event_Type et, Object *obj, Any_Type arg)
{
    char line[1000];
    Call_Private_Data *priv;
    Call *c = (Call*) obj;
    BURST *b;

    Conn *con = c->conn;

    if(strncmp(con->line.iov_base,"REDIR-URL: ",11) == 0)
    {
        priv = CALL_PRIVATE_DATA (c);
        b = priv -> burst;

        char *p = con->line.iov_base;
p+=11;
REQ *req = new_request( strdup(p));
REQ *temp = b->cur_req;
req->next = b->cur_req; // in fact cur_req is the next one
b->cur_req = req;

        // check the burst:
        REQ *curreq = b->req_list;
        while(curreq)
        {
            if (curreq->next == temp) {
                curreq->next = req;
                break;
            }
            curreq = curreq->next;
        }

        else if(strncmp(con->line.iov_base,"REDIR-SERVER: ",14) == 0)
{  // we already created the request
    priv = CALL_PRIVATE_DATA (c);
b = priv -> burst;
    strcpy(line,con->line.iov_base);
    char *p1 = & line[0];
p1+=14;
    char *p2 = strchr(p1,':');
    *p2 = 0;
p2++;
b->cur_req->port = atoi (p2);
    strcpy(b->cur_req->server,p1);
}
}

static void conn_failed(Event_Type et, Object *obj, Any_Type arg)
{
    BURST *b = (BURST*) obj;
burst_failure(b);
}

static void conn_timeout(Event_Type et, Object *obj, Any_Type arg)
{
    BURST *b = (BURST*) obj;
burst_failure(b);
}

static void conn_connected (Event_Type et, Object *obj, Any_Type regarg, Any_Type callarg)
{
    Conn_Private_Data *cpriv;
    Conn *conn;
    Call *c;

    assert (et == EV_CONN_CONNECTED && object_is_conn (obj));

    conn = (Conn *) obj;
cpriv = CONN_PRIVATE_DATA (conn);
    c = cpriv->call;

    core_send (conn, c);
}

static int
burst_create (Any_Type arg)
{

    BURST *b;
    if (num_bursts_generated++ >= param.wsesslog.num_sessions || cur_burst == 0)
        return -1;

    b = cur_burst;
    cur_burst = cur_burst -> next;

    //if(cur_burst == 0)
```c
//

event_signal(EV_SESS_NEW,(Object *)b, arg);
issue_call(b);

return 0;
}

static void
call_done (Event_Type et, Object *obj, Any_Type regarg, Any_Type callarg)
{
    Conn *conn;
    Call *call;
    Call_Private_Data *cpriv;

    assert (et == EV_CALL_RECV_STOP && object_is_call (obj));
    call = (Call *) obj;
    conn = call->conn;

    call_dec_ref (call);
    core_close (conn);

    cpriv = CALL_PRIVATE_DATA (call);
    cpriv->req->lifetime = timer_now() - cpriv->req->birth_time;
}

static void
call_destroyed (Event_Type et, Object *obj, Any_Type regarg, Any_Type callarg)
{
    BURST *burst;
    Call_Private_Data *priv;
    Call *call;

    assert (et == EV_CALL_DESTROYED && object_is_call (obj));
    call = (Call *) obj;
    priv = CALL_PRIVATE_DATA(call);

    burst = priv->burst;

    if(burst->cur_req != 0)
        issue_call(burst);
    else {
        //burst finished
        Any_Type arg;
        arg.l = 0;
        num_bursts_destroyed++;
        event_signal(EV_SESS_DESTROYED,(Object *) burst, arg);
    }

    if (num_bursts_destroyed >= param.wsesslog.num_sessions)
        core_exit();
}

/* Read in session-defining configuration file and create in-memory
   data structures from which to assign uri_s to calls. */
static void
parse_config (void)
{
    FILE *fp;
```
int lineno, i, reqnum;
char line[10000]; /* some uri's get pretty long */
char uri[10000]; /* some uri's get pretty long */
char method_str[1000];
char this_arg[10000];
char contents[10000];
double think_time;
int bytes_read;
REQ *reqptr, *current_req;
BURST *bptr, *current_burst = 0;
char *from, *to, *parsed_so_far;
int ch;
int single_quoted, double_quoted, escaped, done;

fp = fopen (param.wsesslog.file, "r");
if (fp == NULL)
    panic ("%s: can't open %s\n", prog_name, param.wsesslog.file);

for (lineno = 1; fgets (line, sizeof (line), fp); lineno++)
{
    if (line[0] == '#')
        continue; /* skip over comment lines */
    if (sscanf (line, "%s%n", uri, &bytes_read) != 1)
        // Now we only support bursts, not sessions, so ignore the empty
        continue;
    /* looks like a request-specifying line */
    reqptr = new_request (strdup (uri));
    if (current_burst == NULL)
    {
        first_burst = current_burst = new_burst (reqptr);
        current_req = reqptr;
        total_num_bursts++;
    }
    else
    {
        if (!isspace (line[0])) {
            /* this uri starts a new burst */
            current_burst = (current_burst->next = new_burst (reqptr));
            current_req = reqptr;
            total_num_bursts++;
        }
        else
            current_req = (current_req->next = reqptr);
    }
    /* do some common steps for all new requests */
    current_burst->num_reqs++;

    /* parse rest of line to specify additional parameters of this
    request and burst */
    parsed_so_far = line + bytes_read;
    while (sscanf (parsed_so_far, "%s%n", this_arg, &bytes_read) == 1)
```c
if (sscanf (this_arg, "method=%s", method_str) == 1)
{
    for (i = 0; i < HM_LEN; i++)
    {
        if (!strncmp (method_str, call_method_name[i],
                      strlen (call_method_name[i])))
        {
            reqptr->method = i;
            break;
        }
    }
    if (i == HM_LEN)
        panic ("%s: did not recognize method '%s' in %s\n",
               prog_name, method_str, param.wsesslog.file);
}
else if (sscanf (this_arg, "think=%lf", &think_time) == 1)
    ;//current_burst->user_think_time = think_time;
else if (sscanf (this_arg, "contents=%s", contents) == 1)
{
    /* this is tricky since contents might be a quoted
       string with embedded spaces or escaped quotes. We
       should parse this carefully from parsed_so_far */
    from = strchr (parsed_so_far, '=' + 1);
    to = contents;
    single_quoted = FALSE;
    double_quoted = FALSE;
    escaped = FALSE;
    done = FALSE;
    while (((ch = *from++) != '\0' && !done)
    {
        if (escaped == TRUE)
        {
            switch (ch)
            {
            case 'n':
                *to++ = '\n';
                break;
            case 'r':
                *to++ = '\r';
                break;
            case 't':
                *to++ = '\t';
                break;
            case '\n':
                *to++ = '\n';
                /* this allows an escaped newline to
                   continue the parsing to the next line. */
                if (fgets(line, sizeof(line), fp) == NULL)
                {
                    lineno++;
                    panic ("%s: premature EOF seen in '%s'\n",
                            prog_name, param.wsesslog.file);
                }
                parsed_so_far = from = line;
                break;
            default:
                *to++ = ch;
                break;
            }
```
escaped = FALSE;

else if (ch == '\n' & double_quoted)
    { double_quoted = FALSE;

else if (ch == '\' & single_quoted)
    { single_quoted = FALSE;

else

    switch (ch)
    {
    case '\t':
    case '\n':
    case '\':
        if (single_quoted == FALSE &
            double_quoted == FALSE)
            done = TRUE;  /* we are done */
        else
            *to++ = ch;
        break;
    case '\\':               /* backslash */
        escaped = TRUE;
        break;
    case '"':               /* double quote */
        if (single_quoted)
            *to++ = ch;
        else
            double_quoted = TRUE;
        break;
    case '\':               /* single quote */
        if (double_quoted)
            *to++ = ch;
        else
            single_quoted = TRUE;
        break;
    default:
        *to++ = ch;
        break;
    }

    *to = '\0';
else
{
    /* do not recognize this arg */
    panic("%s: did not recognize arg '%s' in %s\n",
        prog_name, this_arg, param.wsesslog.file);
}
parsed_so_far += bytes_read;
}
fclose(fp);
if (DBG > 3)
{
    fprintf(stderr,"%s: session list follows:\n\n", prog_name);
    for (i = 0; /*i < num_templates*/ i==0; i++)
    {
        for (bptr = first_burst; bptr; bptr = bptr->next)
        {
            for (reqptr = bptr->req_list, reqnum = 0;
                reqptr;
                reqptr = reqptr->next, reqnum++)
            {
                if (reqnum >= bptr->num_reqs)
                    panic("%s: internal error detected in parsing %s\n",
                        prog_name, param.wsesslog.file);
                if (reqnum > 0)
                    fprintf(stderr, "\t");
                fprintf(stderr, "%s", reqptr->uri);
                if (reqptr->method != HM_GET)
                    fprintf(stderr, " method=%s",
                        call_method_name[reqptr->method]);
                if (reqptr->contents != NULL)
                    fprintf(stderr, " contents='%s'", reqptr->contents);
                fprintf(stderr, "\n");
            }
        }
    }
    fprintf(stderr, "\n");
}
}
static void
init (void)
{
    Any_Type arg;
    conn_private_data_offset = object_expand (OBJ_CONN, sizeof(Conn_Private_Data));
call_private_data_offset = object_expand (OBJ_CALL, sizeof(Call_Private_Data));
wsesslog_call_priv_offset = call_private_data_offset;
parse_config();
cur_burst = first_burst;
rg_sess.rate = &param.rate;
rg_sess.tick = burst_create;
rg_sess.arg.l = 0;
arg.l = 0;

event_register_handler(EV_CALL_DESTROYED, call_destroyed, arg);

event_register_handler(EV_CONN_CONNECTED, conn_connected, arg);
event_register_handler(EV_CALL_RECV_STOP, call_done, arg);

event_register_handler(EV_CALL_RECV_HDR, call_rcv_header, arg);
event_register_handler(EV_CONN_FAILED, conn_failed, arg);
event_register_handler(EV_CONN_TIMEOUT, conn_timeout, arg);

}
static void
start (void)
{
  rate_generator_start(&rg_sess, EV_SESS_DESTROYED);
}

Load_Generator wsesslog =
{
  "creates log-based session workload",
  init,
  start,
  no_op
};

Table B.3 – sess_stat.c

```c
#include <assert.h>
#include <float.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <httpperf.h>
#include <call.h>
#include <event.h>
#include <session.h>
#include <stats.h>

#include <wsesslog.h>

// As defined in wsesslog.c (calls are created there)
define CALLPRIVATE DATA(c) 
  ((Call_Private_Data *)(char *)(c) + call_private_data_offset))

typedef struct Call_Private_Data
{
  BURT *burst;
```
static size_t call_private_data_offset = -1;

extern size_t wsesslog_call_priv_offset;

extern BURST* first_burst;

static struct {
   u_int num_completed;
   Time lifetime_sum;

   //size_t total_recv_bytes;
} st;

static void perf_sample (Event_Type et, Object *obj, Any_Type reg_arg, Any_Type call_arg) {
}

static void sess_created (Event_Type et, Object *obj, Any_Type regarg) {
   BURST *b = (BURST*) obj;
   b->birth_time = timer_now();
}

static void sess_destroyed (Event_Type et, Object *obj, Any_Type regarg) {
   BURST *b = (BURST*) obj;
   b->lifetime = timer_now() - b->birth_time;
   st.lifetime_sum += b->lifetime;
   st.num_completed++;
}

static void recv_stop (Event_Type et, Object *obj, Any_Type reg_arg, Any_Type call_arg) {
   Call *c = (Call *) obj;
   Conn * con;
   assert (et == EV_CALL_RECV_STOP && object_is_call (c));

   con = c->conn;
   // we don’t want to count data received from peers
   //if ( con-> port < 20000)
   // st.totalRecvBytes += c->reply.header_bytes + c->reply.content_bytes +
   // c->reply.footer_bytes;
static void init (void)
{
    Any_Type arg;
    call_private_data_offset = wsesslog_call_priv_offset;
    arg.l = 0;
    event_register_handler (EV_PERF_SAMPLE, perf_sample, arg);
    event_register_handler (EV_SESS_NEW, sess_created, arg);
    event_register_handler (EV_SESS_DESTROYED, sess_destroyed, arg);
    event_register_handler (EV_CALL_RECV_STOP, recv_stop, arg);
}

static void dump (void)
{
    double min, avg, stddev, delta;
    int i;
    BURST *bur;
    REQ *req;
    i = 0;
    avg = 0.0;
    if (st.num_completed > 0)
    {
        avg = st.lifetime_sum/st.num_completed;
        printf("No bursts %d\n",st.num_completed);
        printf("Avg Burst lifetime [s]: %.3f\n", avg);
    }
    printf("Full details:\n");
    bur = first_burst;
    while (bur && i < param.wsesslog.num_sessions)
    {
        req = bur->req_list;
        while (req)
        {
            printf("%s %s %.3f\n",req->uri,req->server,req->lifetime);
            req = req->next;
        }
        printf("Total = %.3f\n\n",bur->lifetime);
        bur = bur->next;
        i++;
    }
    //printf("Total size of data received from server in Bytes: %d\n",st.total_recv_bytes);
}

Stat_Collector session_stat =
{
    "collects session-related statistics",
    init,
    no_op,
    no_op,
dump
};
REFERENCES


[29] Netcraft February 2010 Web Server Survey, URL:


[30] Ipoque Internet Study 2008/2009, URL:


[35] Content Analysis for the Web 2.0, URL:

[36] Gotocode Forum Application, URL:


[38] Netfilter/Iptables Project HomePage, URL:


[51] P2psim: a simulator for peer-to-peer protocols, URL:


