Performance Evaluation of URL Routing for Content Distribution Networks

by

Zornitza Genova Prodanoff

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Kenneth J. Christensen, Ph.D.
Murali Varanasi, Ph.D.
Rafael Perez, Ph.D.
Kaushal Chari, Ph.D.
Miguel Labrador, Ph.D.

Date of Approval:
July 18, 2003

Keywords: Hyper Text Transfer Protocol, World Wide Web, routing table compression, self-adjusting hash tables, queueing systems

Acknowledgements

I would like to express my gratitude to my advisor Dr. Kenneth J. Christensen for his help and incredible patience. He provided me with valuable insights and advice throughout my Ph.D. studies. He has also spent many hours working on improvements of this dissertation. I would like to thank my committee: Dr. Murali Varanasi, Dr. Rafael Perez, Dr. Kaushal Chari, and Dr. Miguel Labrador. I would also like to thank Loraine Christensen for spending the time to help me improve the quality of the final draft, and Dr. Stephen Suen for his time and comments. And last, but not least, I would like to acknowledge the unconditional support of my family, who are the very source of my inspiration.

Table of Contents

List of Tables iii

List of Figures iv

Abstract vi

Chapter One: Introduction 1
- 1.1 Background 1
- 1.2 Motivation 2
- 1.3 Contributions 3
- 1.4 Organization of this dissertation 3

Chapter Two: Literature Review 5
- 2.1 Basic model of computer network communications 5
- 2.2 Packet switching 8
- 2.3 The Internet and the Web 10
- 2.4 Capability of servers 12
- 2.5 Mechanisms used within CDNs 15
- 2.6 Caching on the Web 16
  - 2.6.1 Sharing of cache directories 18
- 2.7 Application layer routing 24
  - 2.7.1 Redirection at the IP layer 24
  - 2.7.2 Domain name redirection 25
  - 2.7.3 Redirection at the server side 26
  - 2.7.4 Redirection at the client side 26
- 2.8 Hashing algorithms 27
  - 2.8.1 Handling of hashing collisions 27
  - 2.8.2 Uniform hashes 27
  - 2.8.3 Open hash table methods 28
  - 2.8.4 Self-organizing hashing methods 30
    - 2.8.4.1 H1 hashing 30
    - 2.8.4.2 The performance of self-organizing hashes 31

Chapter Three: Scaling from Local Clusters to Distributed CDNs 34
- 3.1 Geographically distributed server clusters 34
- 3.2 Building distributed content 35
  - 3.2.1 Implementing local server clusters 36

3.2.2 Implementing distributed server clusters 37
3.3 Adding content knowledge to server selection 38

Chapter Four: System Design of A URL Router 40
4.1 Operation of a URL router 41
4.2 Architecture of a URL router 43
4.3 Structure of a URL routing table 44
4.4 Performance bottlenecks for a URL router 44

Chapter Five: Using Signatures for URL Routing 46
5.1 Using CRC32 for URL signatures and digesting 46
5.1.1 Handling of collisions 50
5.2 Self-adjusting hashing method for fast table look-ups 51

Chapter Six: Evaluation of URL Signatures for Digesting 54
6.1 Access lists used in the evaluation 54
6.2 Evaluation of CRC32 digesting 56
6.2.1 Experiment results 58
6.3 Comparing CRC32 to MD5-Bloom filter digesting 61
6.3.1 Experiment results 62
6.4 Discussion of results 64

Chapter Seven: Evaluation of Hashing Algorithms for URL Routing 66
7.1 Access lists used in this evaluation 66
7.2 Description of evaluation method – look-up time 67
7.3 Description of evaluation method – queueing behavior 69
7.4 Evaluation of look-up time 72
7.4.1 Experiment results 73
7.5 Evaluation of queueing behavior 75
7.5.1 Experiment results 77
7.6 Discussion of results 80

Chapter Eight: Summary and Directions for Future Research 82
8.1 Specific contributions from this research 84
8.2 Directions for future research 85

References 87

Appendices 93
Appendix A: Source Code for Hashing Algorithms 94

About the Author End Page

List of Tables

Table 2.1 – A comparison of Summary Cache and Cache Digests 25

Table 5.1 – Commonly used CRC polynomials 47

Table 6.1 – Summary statistics for access logs used in this evaluation 58

Table 6.2 – CRC32 URL list collision probability 59

Table 6.3 – CPU Time to generate compressed URL list 59

Table 6.4 – Summary statistics of URL lists used in this evaluation 61

Table 6.5 – Results from experiments #1 and #2 for URL lists CA *net and CSE 63

Table 6.6 – Results from experiments #1 and #2 for URL lists NLANR and VTech 63

Table 7.1 – Characterization of HTTP access lists for size 67

Table 7.2 – Characterization of HTTP access lists for time 67

Table 7.3 – CRC32 URL signatures populated hash table for NASA (K = 8) 73

Table 7.4 – CRC32 URL signatures populated hash table for Clark (K = 8) 73

Table 7.5 – Mean queue length results for experiment #4 80

Table 7.6 – Mean queue length results for experiment #5 80

List of Figures

Figure 2.1 – Functional layers of communication 6

Figure 2.2 – A packet switch network with alternate routes 7

Figure 2.3 – The basic implementation of a communication network 8

Figure 2.4 – TCP/IP layered model and packet structure 9

Figure 2.5 – A sample HTTP GET request header 13

Figure 2.6 – NAT translation 14

Figure 2.7 – IP aliasing 15

Figure 2.8 – A CDN overlay directing an HTTP request to the best content source 16

Figure 2.9 – A Bloom filter 19

Figure 2.10 – Tree-like structure of URL lists 21

Figure 2.11 – Summary Cache: a false hit is redirected after time out 23

Figure 2.12 – Adaptive Web Caching: a false hit is redirected twice 23

Figure 2.13 – Hashing collision of records r_2 and r_{n-1} 28

Figure 2.14 – Chained hashing resolution of collision in Figure 2.13 29

Figure 4.1 – Placement of URL routers in the Internet 41

Figure 4.2 – URL router forwarding or redirecting a request 42

Figure 4.3 – A one-armed URL router 44

Figure 4.4 – The structure of a URL routing table 45

Figure 5.1 – 32-bit multiply algorithm for 64-bit product (from [59]) 50

Figure 5.2 – Hashing algorithms (based on [37]) 53

Figure 6.1 – Snippet of a “raw” HTTP access log 55

Figure 6.2 – Snippet of an URL access list created from the log snippet of Figure 6.1 55

Figure 6.3 – Snippet of an URL list from an HTTP access list showing full URLs 56

Figure 6.4 – The URL list of Figure 6.3 with CRC32 URL signatures 56

Figure 6.5 – The effects of hash table size on look-up time 60

Figure 6.6 – Results from experiment #3 64

Figure 7.1 – Single-server queueing model for hashing algorithm evaluation 70

Figure 7.2 – Hash table look-up time for experiment #1 74

Figure 7.3 – Hash table look-up time for experiment #2 74

Figure 7.4 – Autocorrelation for 100 lags for number of accesses 76

Figure 7.5 – The effect of hash table size (2^K) on mean queue length (L) 77

Figure 7.6 – The effect of burstiness ($T_{\max}$) on L for K=8 78

Figure 7.7 – The effect of $T_{\max}$ on L for K=8 and U=80% 79

Performance Evaluation of URL Routing for Content Distribution Networks

Zornitza Genova Prodanoff

Abstract

As the World Wide Web continues to grow in size, content is being co-located throughout the world in Content Distribution Networks (CDNs). These CDNs need entirely new methods of distributing client requests. The idea of a URL router has been introduced and in this dissertation the performance of URL routing is addressed. A URL router that uses HTTP redirection to automatically forward requests is architected. Significant open problems are finding effective ways to 1) reduce the size of the routing table, and 2) perform fast routing look-ups. These two problems must be solved in order for CDNs to become fully viable and thus are of significant interest.

The first problem is solved by proposing the use of CRC32 as a means of reducing the length of a URL (typically 50 bytes) to a four-byte signature. The CRC32 method is shown to require less CPU resources, generate equal or smaller size digests, achieve equal collision rates, and simplify switching when compared to existing MD5-Bloom and CRC19 methods.

The second open problem is solved by a new “Aggressive” hashing algorithm. The average look-up time performance of Aggressive hashing is compared to that of simple

chain hashing and self-adjusting H1 hashing. Aggressive hashing outperforms in look-up time simple chain hashing by a factor of 16 and H1 by a factor of about two.

A major contribution of this dissertation is studying the queueing behavior of hashing algorithms. A simulation model was designed as a single server queueing system where the arrival rate is determined by an HTTP trace and the service time by hash table look up. For a constant throughput, the new Aggressive hashing method exhibits 27 times shorter mean queue length than simple chain hashing and about two times shorter than H1 hashing. For constant utilization, the results were surprising in that H1 and Aggressive result in several magnitudes difference in mean queue length. The isolated cause for this counter-intuitive result is the high level of autocorrelation in the H1 hashing look-up time. In conclusion, the CRC32 URL signature method and Aggressive hashing algorithm investigated in this dissertation are better than existing methods for designing the next generation of URL routers to enable future growth of the Internet.

Chapter One

Introduction

1.1 Background

The distribution of content throughout the Internet via large-scale Content Distribution Networks (CDNs) is becoming more and more prevalent. A CDN is an invisible overlay of hardware and software components on the Internet. In a CDN, a single origin server site contains the "original" content. This content is then partially co-located throughout the Internet in content sources, such as distributed servers and caches. For example, content originating in California, but accessed frequently in Florida, can be automatically mirrored in a content server owned by the CDN service provider in Florida. This content mirroring 1) reduces the load on the origin server, 2) reduces traffic on the Internet, and 3) improves response time to the users. For CDNs to be feasible, methods of routing HTTP requests originating from users, or clients, are needed. URL routers are one way of routing – or redirecting – HTTP requests from an origin server to some other content source in the CDN. A large overhead is required to maintain state information about the distributed content and to determine to which content source a client should be directed.

Akamai [1] is an example of a commercial CDN. Akamai is the world's largest distributed network, consisting of more than 15,000 servers in over 60 countries forming more than 1,100 networks. Akamai directs requests to geographically distributed and co-located content as follows. An incoming HTTP request to an origin server is replied to

with the HTML main page. A typical HTML page contains embedded links for images and other document components. These embedded links are modified by the Akamai service running at the origin site to point to temporary servers nearer to the client than is the origin server. Thus, the origin server always sends the HTML main page, and the distributed servers send the embedded images. The embedded links are said to be "Akamaized". The Internet uses the IP protocol to transfer data in the form of packets between server and clients. An HTTP request contains a URL string, and the HTTP response contains the requested object (HTML or image file). A CDN contains mechanisms and policies for content serving, distribution, and request routing. CDNs are usually commercial enterprises [33], [31], [62] that host content for content providers. For example, Yahoo [78] and CNN [10] use Akamai CDN services. Content is created at one or more central sites and located on the origin server. Distribution from the origin server into content sources owned by the CDN provider occurs automatically by either "pushing" (that is to distributed servers) or "pulling" (that is into reverse, transparent, and proxy caches, described later in this section). In this dissertation, the distribution of content into CDN owned content sources is not addressed.

1.2 Motivation

In his speech at the Telecom 1999 conference, John Roth, chief executive of Nortel Networks said that 2.5 billion hours were spent waiting on the Web in 1998 [55]. This extreme delay in the Internet is caused by the inability to efficiently distribute and access content in the Web. To solve the problem, it is necessary to coordinate between the content sources within a single CDN to ensure that the state of content is known. The key

function of a CDN overlaid on top of the Web is to force each client request to be serviced by the most appropriate content source. This entails knowledge of location and state of content sources, location of the client sending a request, and available paths between the client and all content sources.

This dissertation addresses the problems of reducing the size of shared routing tables and performing fast look-ups in routing tables. Solving these problems will help reduce the load on the CDN origin server, reduce traffic on the Internet, and ultimately reduce response time to users and wait time on the Web.

1.3 Contributions of this dissertation

This dissertation investigates new methods related to CDNs, in particular to their implementation of application layer routing. It also provides empirical evaluation of routing table look-up (hashing) methods. The main contributions of this work are:

- Architected a new URL router that uses HTTP redirection
- Investigated new use of CRC32 for reducing the size of routing tables
- Investigated a new self-adjusting hashing method for faster URL routing look-up
- Performed the first queuing evaluation of hashing where the effects of correlation were discovered for the first time

1.4 Organization of this dissertation

The remainder of this dissertation is organized as follows:

- Chapter Two reviews basic concepts in packet switching, routing, HTTP, server clusters, and mechanisms used in CDNs. This chapter also reviews current

literature in caching, server load balancing, application-layer routing, hashing algorithms as applicable to the research in this dissertation.

- Chapter Three describes how server selection methods in a CDN differ from those in local server clusters due to bandwidth and delay constraints in the Internet. This chapter describes several server selection criteria and outlines how caching information can be used in server selection.
- Chapter Four architects a URL router for a CDN. The system design of a URL router is presented.
- In Chapter Five CRC32 signature generation methods are described and their application to signature-based digesting for URL routers is presented. A new Aggressive hashing algorithm is also presented that can be used for fast look-up in URL routing tables.
- Chapter Six evaluates and compares the URL routing look-up performance of CRC32 digests to MD5-Bloom digests and other digesting methods from the literature. The evaluation uses trace workloads.
- Chapter Seven evaluates and compares the performance of three hashing methods – including the new Aggressive hashing – for URL routing. This evaluation measures the hashing table look-up time and the behavior of a single server queue, where the service center is the hashing table look-up. The evaluation uses trace workloads.
- Chapter Eight summarizes this work and describes possible directions for future research.

Chapter Two

Literature Review

This chapter reviews basic concepts in packet switching, routing, HTTP, server clusters, and mechanisms used in CDNs. It also reviews current literature in caching, digesting methods used in caching infrastructures, load balancing in server clusters, and application layer routing. In addition, this chapter provides an overview of existing work in hashing algorithms as applicable to this research.

2.1 Basic model of computer network communications

Communication on the Web is achieved by way of the general principles of computer networks. By enforcing a reference communication model, and associated standards, any two computers are enabled to *connect*, or be ready to exchange information. Computer communication is subdivided into functional layers. A complete set of rules, known as a *protocol*, governs information exchange between same-level layers. Each layer performs unique and specific tasks, and uses the services of the layer below it, while at the same time providing services to the layer above it. Layers can be viewed as "black boxes" or independent units. Providing a thorough introduction to some specific layered models, such as the Open Systems Interconnect (OSI) model [11], is beyond the scope of this work. In this dissertation, the focus is on the TCP/IP [17][74] and HTTP [21] protocols

(introduced later in this chapter) and their corresponding layers, since TCP/IP is the underlying protocol of the Internet and TCP/IP and HTTP are both protocols of the Web.

Figure 2.1 shows how application data is transferred between two computers that are connected to a network. Software, called a "protocol stack", is running on both computers. Data format is changed at each layer at the sender "down the stack" and some information (overhead) is appended to the data to form a unit of transfer. After being processed by the lowest layer, the data are sent on what is known as the transmission medium. At the other end, the data undergo the reverse procedure of "climbing up the stack" while finally being converted back into the format suitable to applications.

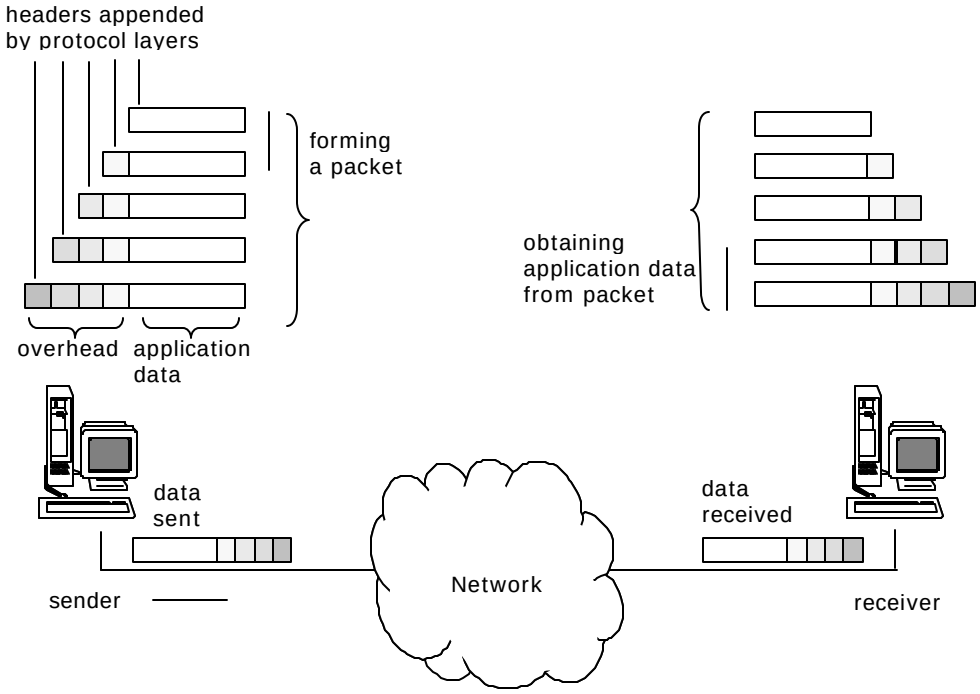

Figure 2.1 – Functional layers of communication

The layer mode of Figure 2.1 is based on the concept of interfacing. An interface is a complete set of rules for information exchange between consecutive layers at the same

site (or computer location). Interfaces eliminate the need for upper layers to include provisions about lower layer protocol details.

Networks are often represented as graphs, with computers being nodes in the graphs, and edges being communication links. Figure 2.2 depicts a network with alternate paths, called *routes*, between nodes 1 and 5. This graph shows how packets can be forwarded to the destination host through several alternate paths.

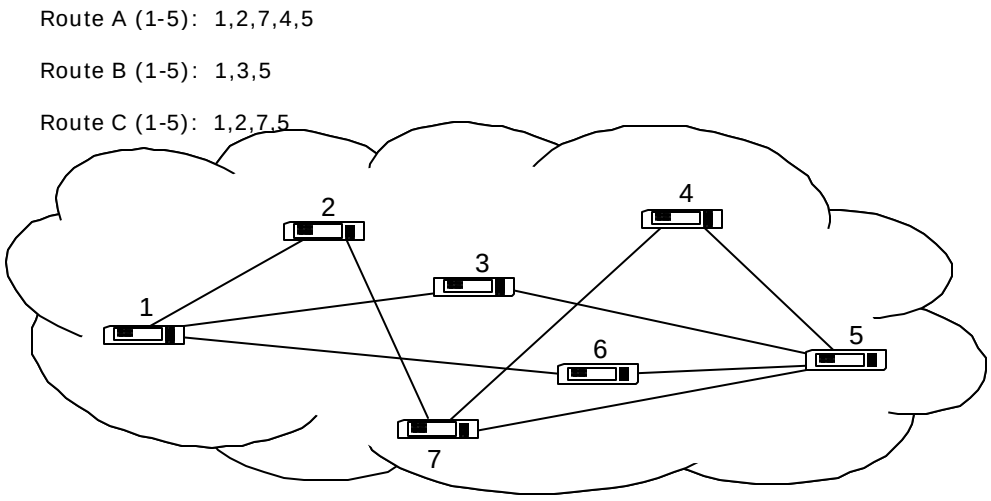

Figure 2.2 – A packet switch network with alternate routes

A network *host* is a computer with CPU, memory, and I/O devices that are interconnected by a back plane bus as shown in Figure 2.3. Network *adapters* are one example of an interface implementation. Adapters are sometimes called Network Interface Cards (NIC). These are I/O devices responsible for transferring data from a channel (or medium) to the computer and vice versa, that is, convert the data between

two formats: signal on bus and signal on network medium. A workstation with an Ethernet (IEEE 802.3) [54] adapter is one example of a host.

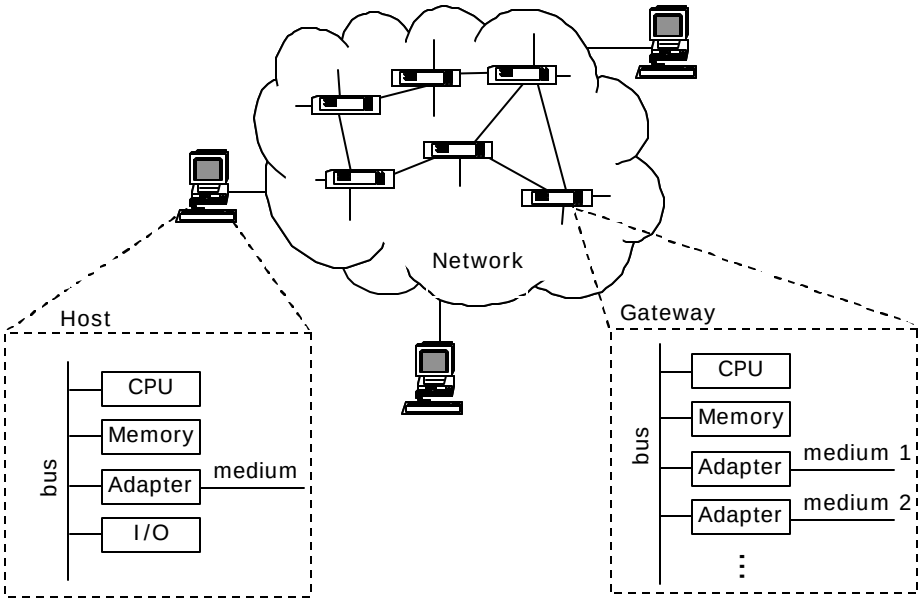

Figure 2.3 – The basic implementation of a communication network

2.2 Packet switching

The Internet is a packet switching type of network. In a packet switching network, transmitted messages are segmented into units, called *packets.* A packet can be viewed as a string of bits that is transmitted serially over a wired or wireless network medium, known as a *link*, that connects two computers. Figure 2.4 depicts the basic structure of a packet. Application bits are encapsulated by *header* and *trailer* sub-strings. Each protocol layer [70] appends a corresponding sequence of header and/or trailer bits at packet creation. The *link layer* (also called the data link layer) is responsible for interfacing with the medium. The *network layer* is responsible for moving the packets

across a single network or multiple networks, and specifies the protocol for *hop-to-hop* delivery. All gateways require an implementation of the network layer. The transport and application layers provide for end-to-end communication, and only hosts must implement those layers.

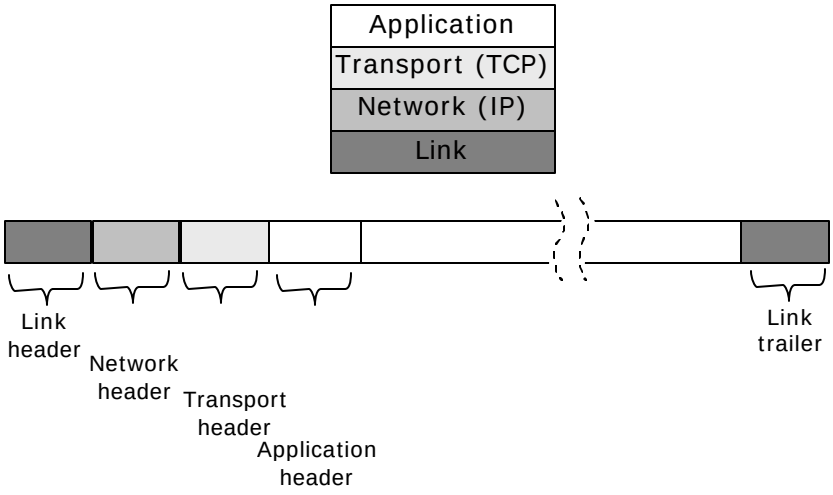

Figure 2.4 – TCP/IP layered model and packet structure

Individual packets are sent one by one over the network links. When all message packets arrive at their destination, they are reassembled into the original message and made available to applications. To take advantage of resource sharing, packets share link and node resources. Each node, whether a host or a gateway, has one or more memory buffers, called *outgoing queues,* that are modeled as queues with a single server facility. Each queue-server facility pair can be viewed as being designated to one of the network links going into the node. Packets are created at a node and placed in a corresponding queue. At this time, they are given an end destination by including a destination *IP address* (a 32 bit value for IPv4 [17]) in the packet IP header. When a link resource

(associated with some queue) becomes available, the head packet is forwarded, and is thus removed from the queue, and the next packet "in line" moves to the head of the queue.

Gateways act as *bridges* or *routers*. Bridges operate at the link layer and make multiple (same-type) networks appear as a single net for upper layers. Bridges can detect and discard corrupted packets; they do not modify packet bits. Routers are computers that connect two or more (dissimilar) networks together. They are capable of receiving a packet on one of the networks and then transmitting it across another network. Routers must modify lower layer packet header bits in order to route a packet from router to router, and finally to the destination computer. Routers operate at the network layer; they have an adapter for each network to which they are attached and they can connect networks of different types. TCP/IP networks are built by using routers rather than bridges.

Routing on the Internet at the IP-layer is currently done with IPv4 addresses. The Routing Information Protocol (RIP) [28] sends routing-update messages at regular intervals and when the network topology changes. Thus, RIP is a distance-vector protocol. The Open Shortest Path First (OSPF) protocol [45] uses a routing algorithm that keeps track of link state, where each router maintains the state description of its local links to networks. There are no existent routing protocols for application-level routing (e.g., for URL routers) since application-level routing is very new. Thus, there are no defined mechanisms or policies for building routing tables in URL routers. There is ongoing work to define the mechanisms needed for URL routing protocols [22]. The Internet Content Application Protocol (ICAP) [18] is a protocol for lightweight vectoring

of HTTP services. ICAP servers transform URLs for purposes including routing of HTTP requests. ICAP is a mechanism for modifying HTTP requests. ICAP does not address the routing protocols that determine how a request should be modified. As described in Section 2.6, caching infrastructures have defined protocols (e.g., ICP [76]) to exchange information on cached content. Such protocols may apply to URL routers this is future work that is beyond the scope of this dissertation.

2.3 The Internet and the Web

The Internet is currently the largest network and application of packet switching. It creates the illusion of a single network, but is actually a network of interconnected networks of the same or a different type. At the user level, the Web appears as a "web" of computer files residing at different physical locations and interconnected by hyperlinks, or pointers. These hyperlinks are contained within a Web site (a set of files made available for public access) and can be traversed to reach other Web sites.

The Hypertext Transfer Protocol (HTTP) is the application-layer protocol of the Web. It is used to deliver files, or *resources*. Some examples of resources delivered via HTTP would be ASCII text files, binary executables, HTML files, and image files. HTTP is a "request/response" protocol that operates at the application layer.

In general, a resource is a file(s) that can be identified by a Universal Resource Locator (URL). Resources are considered as either *static* or *dynamic*. For example, an image or a text file is considered a static resource, whereas the output of a Common Gateway Interface (CGI) script would be a dynamic document (CGI scripts are programs

that return a dynamic query result to the client only when the result data conforms to a standard of transfer between the server and the script).

HTTP supports a client server model of communication. A Web browser, like Netscape [48] or Internet Explorer [42], is an example of an HTTP client. It sends requests to a Web server, which then sends responses back to the client. The standard port for HTTP servers to listen on is 80, though they can use any port. An HTTP client opens a connection and sends a request message to an HTTP server; the server then returns a response message, which usually contains the requested resource. After delivering the response, the server closes the connection. Once the connection is closed, all information about the state of the remote machine is lost. HTTP is thus a stateless protocol. Once the client-server connection is dropped, the server retains no history of the communication that took place. *Cookies*, or <attribute, value> pairs, are stored at the client computer and are used to record transaction history that is later used on consecutive connections to the same server. HTTP provides support for client side *caching*. Caching refers to the retention of local copies of previously retrieved content.

HTTP transactions are called methods. Methods are specified in the header of the HTTP request message, which is the first message that gets sent to the server after a TCP connection is established. There are a number of request methods supported by HTTP: GET, HEAD, PUT, POST, DELETE, and TRACE. GET allows the access of a remote resource and is the most commonly used HTTP method. The required fields are shown in line one of Figure 2.5: “GET [URL] HTTP/[version]”. All other fields are optional. The HTTP response format is “HTTP/[version] code”. All other fields, such as date, server type, date last modified, and E-tag (a unique number for each response) are optional.

An example of a GET (client) request header is shown in Figure 2.5 . Web server operation consists of the following steps: wait for a connection, make the connection, receive command and parse it, respond with requested data, and drop the connection. The above steps need to be performed in parallel for simultaneous requests.

```
GET/ index.htm HTTP/1.1
Accept: image/gif, image/x-xbitmap, image.jpeg, image/pjpeg,
application/vnd.ms-powerpoint, application/vnd.ms-excel,
application/msword, */*
Acccept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
Host: 131.247.3.42
Connection: Keep-Alive
```

Figure 2.5 – A sample HTTP GET request header

2.4 Capability of servers

A server cluster is a group of independent computers working together as a single system to ensure that resources remain available to clients. A cluster appears to its clients as a single server, but is actually a group of servers acting as one. A cluster provides better scalability and availability than a single server. New servers (or hardware) can be added to a cluster to increase its capacity: when the load at the server increases, an incremental and linear increase in hardware can make the load increase transparent to the client.

A cluster uses the redundancy of multiple servers to overcome failures. Fault-tolerance is handled in software. Multiple servers in the cluster can provide the same service. Server clusters are built as interconnected workstations, that is, high speed storage area networks (SANs), that enable direct connections between heterogeneous storage servers or Gigabit Ethernet can also be used for workstation interconnection. Server clusters allow for load balancing: content requests or incoming client connections

and content delivery (known as *server load*) can be distributed within the cluster. A front-end computer is part of a server cluster site and handles commonly requested objects without having to burden the server hosts with additional connections. Several well-known techniques for server side HTTP request redirection are described next.

Server-side Domain Name Service (DNS) [44] is used when each server in the cluster is assigned a different valid IP address. Incoming content requests are then redirected in a round robin, thus distributing the load among all servers in the cluster.

NAT, Network Address Translation [57], is a technique used to forward packets between a public and private network via a single device (front end). An example is shown in Figure 2.6. The private network (a server cluster) has IP address range 131.247.3.x of IP addresses. These addresses are only accessible through the front end that has a single valid public IP address, 158.102.1.11. The front end will then have the addresses 131.247.3.1 (private) and 158.102.1.11 (public) and will forward packets between the two networks by rewriting the corresponding IP addresses.

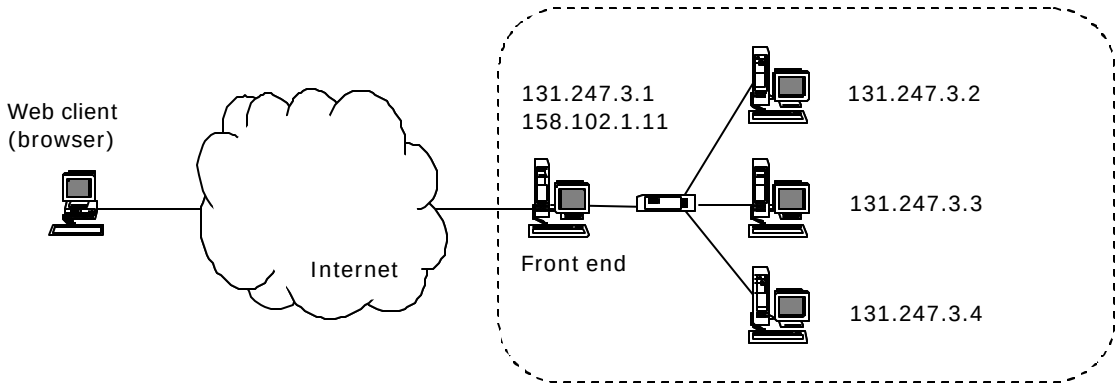

Figure 2.6 – NAT translation

IP aliasing is used when several IP addresses are assigned to a single front end, in which case different Web documents can be served according to the IP addresses (host names) used in the HTTP request message. Figure 2.7 depicts such a case.

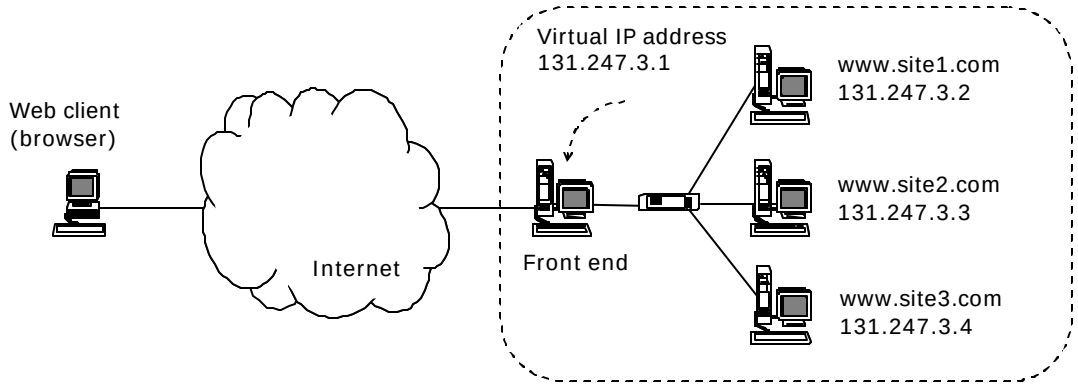

Figure 2.7 – IP aliasing

2.5 Mechanisms used within a CDN

The mechanisms within a CDN work on top of IP routing. These mechanisms are specialized routing capabilities within the Internet and at content sources. Policies are needed at these mechanisms to determine the "best" content source and to employ the appropriate mechanisms for directing requests to this source. The best content source can be defined socially (in terms of reducing traffic on the Internet) or greedily (in terms of reducing the response time as seen by the client). A socially defined content source will result in the least Internet resources consumed per response. Thus, a socially defined content source will likely be the closest source to a client. A greedily defined content source will result in the smallest response time for the client. This may entail consuming more Internet resources than a request serviced by a socially defined source.

Figure 2.8 shows a CDN overlaid on the Internet and a request being directed to a content source other than the origin server; it also shows the origin server as multiple servers at a single site, along with temporary servers, *proxy caches*, and *transparent caches*. Proxy caches are gateways where Web clients are configured to ask the cache for all documents; proxy caches are supported by HTTP, while *transparent caches* intercept HTTP requests on their way from client to server and generate a response as if the response came from the server, when in fact it came from the cache. Even though the proxy cache is geographically closer to the client than it is to the distributed server, the distributed server will satisfy this request better. This may be the case when the proxy cache is heavily loaded, while the distributed server is experiencing less traffic, or if the network conditions between the client and the proxy are unfavorable due to link failure or traffic delays.

CDNs are implemented by using devices that forward or route at layers higher than the IP layer. Such routers can be used to forward or redirect a client HTTP request to the best content source.

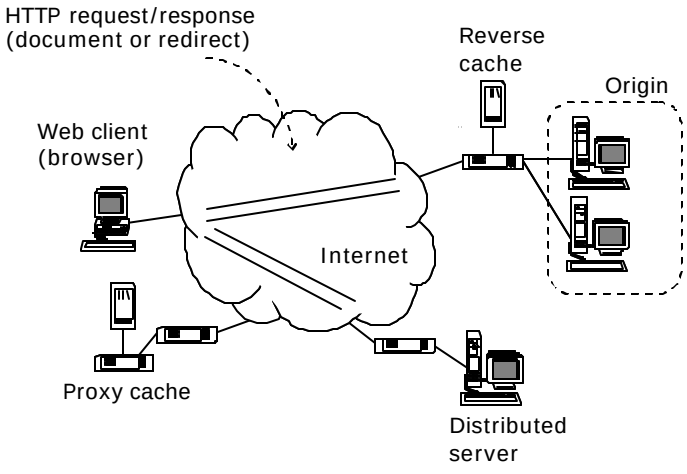

Figure 2.8 – A CDN overlay directing an HTTP request to the best content source

2.6 Caching on the Web

To facilitate faster browser access to globally available content, static Web documents are disseminated, or *cached*, throughout the Internet. In addition, caching can reduce both Internet traffic and server load. Web caching can occur at the client site in a proxy cache, at the origin server site in a reverse cache, or in transparent caching infrastructures within the Internet. When caching is implemented local to the client, it is done in the browser application or via proxy servers in the client LAN. Another caching strategy, transparent caching, would position hierarchies or meshes of caches in the Internet, such as in the case of geographical push-caching [27]. Placing the nodes at the server site refers to reverse caching and is part of a local server site. Reverse caches handle commonly requested objects without having to burden the server hosts with additional connections.

Caching structures use compressed cache directory information, called digests, to coordinate cooperation between caches (that is SQUID – a free Web Proxy software part of some Linux distributions [77]). Caching is largely limited to static content and the use of digests may present performance-scaling problems. An HTTP request sent to a cache may result in a miss, in which case the cache device obtains the requested content from another cache or the origin server (and then responds to the original request). Distributed caching infrastructures share knowledge of cache contents (that is, the currently stored objects in the form of a list of URLs). In this manner, a cache device that does not contain the document being requested, can request that document from another cache device that is known to contain the object.

2.6.1 Sharing of cache directories

Sharing of caches can be implemented by distributing lists of URLs called *cache digests* between caches. A miss at a given cache site can then be forwarded, using the URL lists as a forwarding table, to another cache that contains the requested document. Occasional false hits, or routing collisions, will occur when the URL lists are incorrect because of changes in cache contents that occur between sharing updates for the URL lists. A false hit occurs when either 1) two or more URLs reduce into the same bit pattern, or *signature*, which can result in forwarding a request to the wrong cache device; or 2) the requested object has expired out of the cache to which the request was forwarded, and an update message has not yet been sent by this cache. False hits are handled in the same manner as misses. Signature collisions are expected to be negligible as compared to cache aging rates. A typical cache miss rate is about 5% from the total number of requests [69], while signature collisions are of smaller scale, as shown later in chapter 6.

In Summary Cache [19], each URL in a URL list is hashed into a 128-bit value using an MD5 signature [58],[73]. The 128-bit signatures are then partitioned into four 32-bit quantities and further reduced (by modulo division) to become indexes into a Bloom filter [9] digest. The use of Bloom filters reduces the size of the digests as compared to sending URL lists between caches, the tradeoff being that false hits are introduced as a result of Bloom filter collisions.

The Bloom Filter is implemented as a one dimensional array. Figure 2.9 shows a Bloom filter that stores the key *K*. The array is first initialized to zero. When the key *K* is inserted into the filter a set of *k* hash functions are calculated over it and *k* individual Bloom filter bits are then set to true (a value of true is denoted as “1” on Figure 2.9).

Individual bit positions are selected to be set based on the indexes obtained from the computation of the set of hash functions.

Similar to other hash table structures, the Bloom filter is probabilistic. Two or more keys can hash to the same set of index locations. The major benefit of using a set of function computations to produce an index, versus a single function is a reduction in the number of collisions.

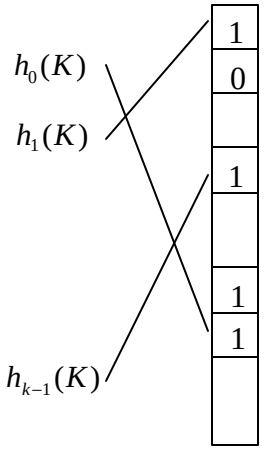

Figure 2.9 – A Bloom filter

Consider two different implementations using an array of a fixed size – a Bloom filter-based and a hash table implementation that employs a single function. The Bloom filter will yield fewer collisions than a hash table that employs any of the hash functions from the function set, if individually taken. With the Bloom filter implementation one needs to set to true a set of bits, instead of a single one; and the probability of collision in the filter is the product of the individual collision probabilities for each function. This product probability is a much smaller value than the individual collision probability of

any function from the set. A dense Bloom filter will have many of its bits set to true, and hence collision probability in a dense filter will be higher than that of a sparsely populated one. The Summary Cache implementation of Bloom filters further reduces the size of the filter before forming a digest by using a number – 8, 16, or 32, called the load factor. The total number of bits in the Bloom filter are 8, 16, and 32 times the average number of unique URLs (or documents) in the cache. This results in introducing additional collisions. For analysis of the performance of Bloom filters refer to [9]. The major trade off is that key deletions are not supported. To remove a key from a single hash function table it is merely needed to unset the bit in the corresponding index position. In a Bloom filter this cannot be done, since the same index position(s) may have been set when more than one key was stored.

To allow for key deletions, the Summary Cache implementation uses a one-dimensional array to build the filter; but instead of setting individual bits, when a key is stored in the array, a four bit counter is incremented. The counter for each position in the corresponding set is incremented when a URL entry is added. This allows for keys to be deleted by decrementing the set counters.

A Bloom filter collision can result in redirecting an HTTP request to the wrong server (content source). A routing collision results in the (mis)selected document source (cache) retrieving the requested document from the origin server. The document will then be available for subsequent requests. Thus a collision in a cache is not a serious problem.

The Summary Cache [19]digesting protocol is overlaid on top of the Internet Cache Protocol ICP [76] and is implemented as a modification of Squid 1.1.4 [77]. The digest contains the local Bloom filter, while an additional array is added to it for each neighbor.

The Adaptive Web Caching (AWC) digesting method [79] is another application-layer cache routing protocol for content dissemination and access. AWC maintains a data structure at each site to keep track of currently available content. In order to maximize cache sharing with nearby positioned cache proxies, AWC distinguishes between proxy and server content.

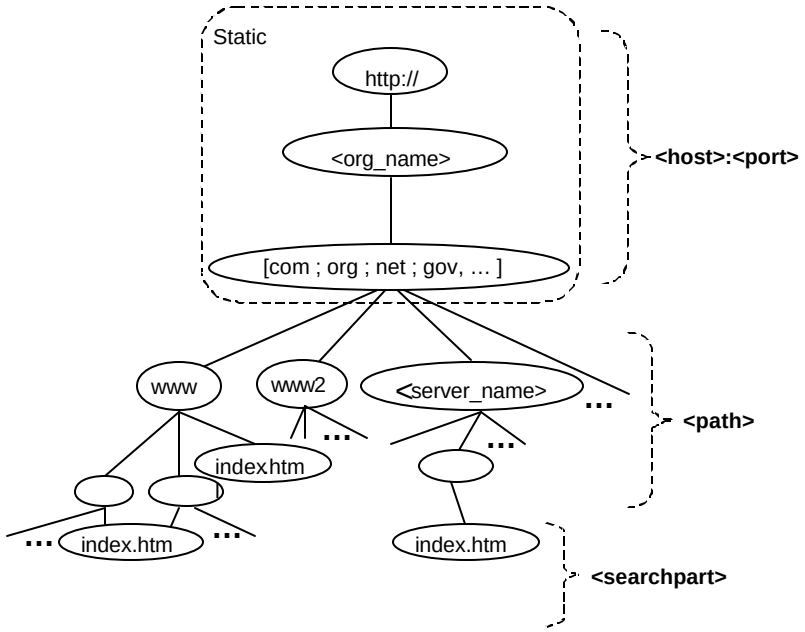

Figure 2.10 – Tree-like structure of URL lists

URL strings are organized into a tree structure. The scheme, host name, path, and file name component of each URL string can be looked at as tree nodes, where the root is the "http://" part of the URL string. Following the edges (from any leaf node up to the root) one can reconstruct a single URL string that belongs to the site (see Figure 2.10); 19-bit CRC signatures of each tree node can then be used form a structure called *an incremental hash chain*. Information is shared, so that each URL is represented as a hash chain

sequence comprised of 19 bit codes for each component, that are coupled together with tree position information. Collisions can still occur for non-unique CRCs.

The size of a URL hash chain is a function of the number of components in a URL. It can be shown that this model does not perform well for wide-base trees, where there is a little overlap between URL components for different URLs.

If the content is not available locally, the requested URL will be forwarded to a cache, which contains a cached copy of the requested document. Figures 2.11 and 2.12 depict a false hit handled by Summary Cache and AWC. When a request arrives at node A for a document *x*, which is not locally available, Summary Cache checks all local bit arrays to determine what sibling caches keep a copy of *x*. A query is sent to cache B due to a false hit in the bit array of B kept at cache A (the only hit for this group). After a timeout, A determines that *x* is not available locally, and forwards the request to the parent node C, which has a copy of *x*. In the same situation, AWC will not send out a query message, but will, rather, forward the request to the sibling node B, which in turn forwards it to the parent node directly, thus avoiding a time out. (Note that AWC implements a mechanism to avoid forwarding loops.)

The Cache Array Routing Protocol (CARP) [75] uses hash-based routing to provide a deterministic "request resolution path" through an array of proxies. The request resolution path means that for any given URL request, the browser, or a nearby proxy server, will know exactly where in the proxy array the information will be stored, whether already cached from a previous request, or making a first Internet hit for delivery and caching.

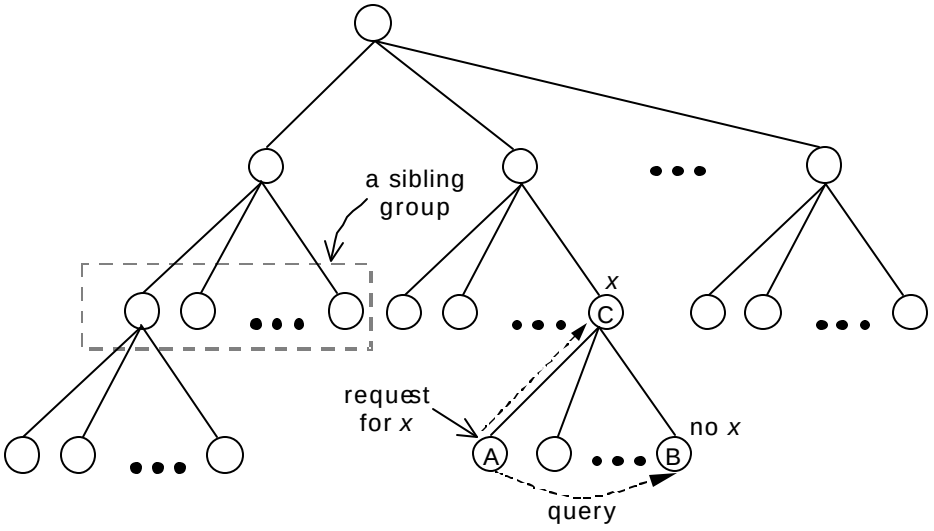

Figure 2.11 – Summary Cache: a false hit is redirected after time out

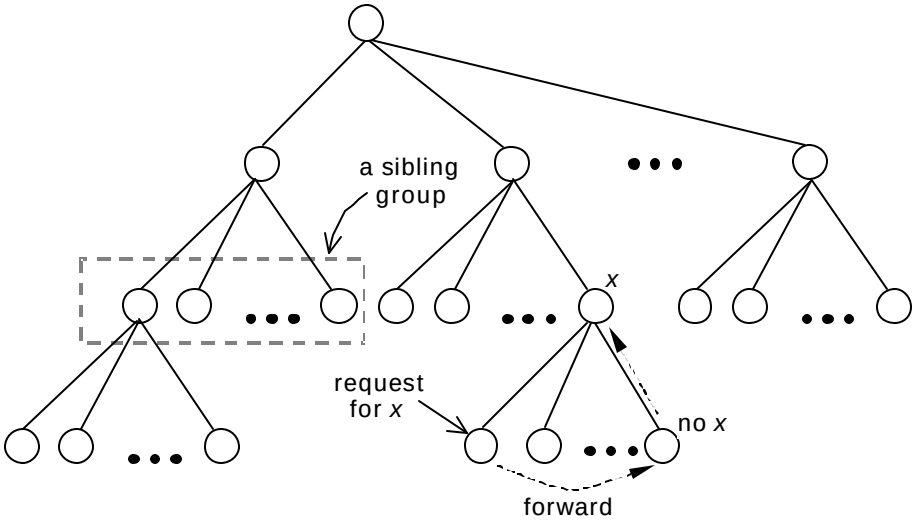

Figure 2.12 – Adaptive Web Caching: a false hit is redirected twice

Because CARP provides a deterministic request resolution path, there is no query messaging between proxy servers (which creates a heavier congestion of queries the

greater the number of servers), as there is with conventional ICP networks. CARP eliminates the duplication of contents that otherwise occurs on an array of proxy servers. With an ICP network, an array of five proxy servers can rapidly evolve into essentially duplicate caches of the most frequently requested URLs. The result is a faster response to queries and a more efficient use of server resources. It is not clear how well the approach performs for wide-area cache sharing, where proxies are distributed within a network.

Cache Digests [60] is a cache routing table compression technique that enables caches to make information about their local cache content available to peers. Instead of posing queries to peers and waiting for replies, *digests* are used to identify co-operating caches that are likely to have a given web object. Digests are similar to routing tables. They are compressed URL lists where each URL is associated with a set of caches that accommodate the content it presents. Table 2.1 presents a comparison between routing table compression and exchange done in Summary Cache versus Cache Digests.

2.7 Application layer routing

Application layer routers are intended to reduce the load on content sources allowing them to serve content only and not to handle routing at the same time. Such routers operate at the application layer and require additional handling of TCP connection semantics and lower layer processing.

2.7.1 Redirection at the IP layer

Anycasting and centralized approaches, such as Global IP-Aanycast [35], attempt to solve server location problems at the IP level. Anycast IP addressing [20], [47] provides a

mechanism for routers or specialized servers to determine the "best" of an available set of servers. Unfortunately, at the IP layer, no information about the state of the selected "best" server can be communicated between the routers and specialized servers; and a request may be forwarded to a server, which is unavailable or has a high response time. Hence, anycast methods are high in overhead or require infrastructure changes.

Table 2.1 – A comparison of Summary Cache and Cache Digests

	Summary Cache	Cache Digests
Extends ICP	Yes	No
Vertical hierarchy of caching proxies	Yes	Yes
Push/pull strategy	Push	Pull
Bloom filters	Yes	Yes
Supports deletion when object is purged from cache	Yes	No
Maintains state of children proxies	Yes	No
Piggybacking of update messages	No	Yes

2.7.2 Domain name redirection

A "hosts.txt" file that listed all hosts in the Internet, supplied the original Internet directory service. As the Internet grew, this approach was replaced by DNS [44] in 1985. Some DNS servers measure round-trip times to known name servers in order to choose the lowest-latency server, especially at the root level. Although this can improve the performance of name lookups by lowering the mean lookup latency, it only helps at one level of a cache miss.

2.7.3 Redirection at the server side

Current wide-area content routing depends on HTTP or DNS-level redirection; and is generally handled on the "server side". Cisco's Distributed Director (DD) [13] redirects a name lookup from the main site to a replica site closer to the requesting client's address, based on responses from a set of participating routers running an agent protocol supporting DD. Proprietary schemes by Akamai, Sightpath, Arrowpoint and others appear to work similarly. These proprietary CDNs can be centrally monitored and managed, unlike name-based routing. This may lead to a better understanding of network performance; however, CDNs rely upon the existing IP routing framework for content delivery, so the amount of benefit to be gained from a proprietary overlay network is limited.

2.7.4 Redirection at the client side

Client-based methods [14] use a ping type approach to find the fastest responding server for a given request. These methods generate excessive and redundant overhead traffic. Smart server selection [72] is one approach to content routing, where, upon a request, a central name server for some Web site provides to the client browser all available addresses for replicas of the content. The client (or the client's DNS server) then probes a nearby router and chooses the nearest (that is, in number of hops) server. This method can only be implemented by making changes in the infrastructure, that is implement a protocol to guide the communication between clients and nearby routers.

2.8 Hashing algorithms

Hashing is an efficient type of searching. A hash table is an array used to store data and make them available for fast access. A key field called the *hash key* uniquely identifies each record, or a row, in the hash table. This key is mapped to a *hash index* by a hash function, that is key x (from the set of all possible keys) is mapped to an index (from the smaller set of finite number positions in the array) by a hash function $h(x)$. Hash functions are not one-to-one; two or more keys could map to the same index. Such incidents are called *collisions*. To guarantee access to data, collisions need to be resolved. A mechanism is needed to make two or more records accessible from within the same hash table location.

2.8.1 Handling of hashing collisions

Let x be a key, such that $x \in (1,..., m-1)$, that is there are m positions in the hash table. Figure 2.13 shows a hash table with collision. The hash indices for k and k_{n-1} are the same, that is $h(k_2) = h(k_{n-1})$. In such a case, both records r_2 and r_{n-1} have to be stored at the same location in the table, which is not possible. This is called a *collision*. Collisions need to be resolved.

2.8.2 Uniform hashes

If the probability that a key x is in the set of keys is $P(x)$, then if there are m slots in the hash table, a uniform hashing function $h(x)$ would ensure:

$$\sum_{h|h(x)=0} P(x) = \sum_{h|h(x)=1} P(x) = ... = \sum_{h|h(x)=m-1} P(x) = \frac{1}{m} \qquad (2.1)$$

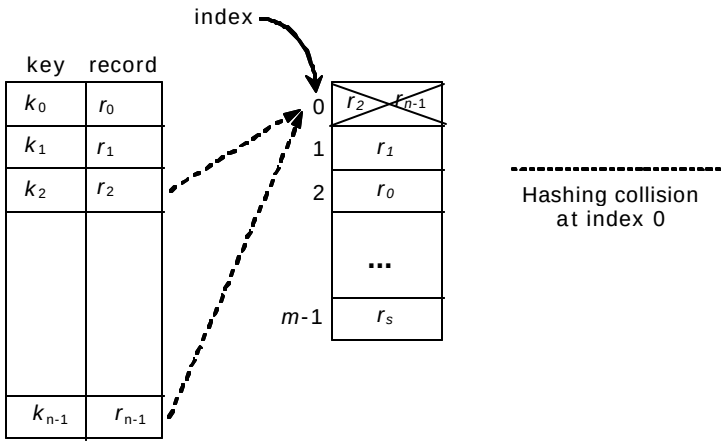

Figure 2.13 – Hashing collision of records r_2 and r_{n-1}

When the keys $x \in (0, r]$ are randomly distributed in interval $(0, r]$, then it is easy to choose a hash function $h(x)$, such that it provides for uniform hashing, that is a function that maps the keys to a sub-string of the key bits can be used and results in less collisions than non-uniform hashing.

2.8.3 Open hash table methods

When resolving collisions, a decision has to be made whether to keep the size of the table fixed, or, if not restricted by space, to extend the hash table, when needed. In the first case two methods are used to resolve collisions: *nonlinear rehashing* (that is generate another hash key) or *linear rehashing* (step through the table until an available slot is found). In the second case, one way of resolving collisions is to build a *chained* hash, or

table such that the original array contains a pointer in each location or slot that points to a linked list structure of hash keys that hash to the same slot in the table.

As shown in Figure 2.14, when records r_2 and r_{n-1} collide, that is when both need to be stored at location *0* in the hash table, one implementation of a chained hashing algorithm will place the first colliding record at the head of a linked list, and later append the second colliding record to the end of the list.

Hash functions should have low collision rates; that is, it should be difficult to make strings that produce the same key. For this reason, such functions must be tailored to the set of keys to be hashed. A common hash function is $h(x) = x \bmod m$, which returns the remainder of key x after division by m. It has been shown that the widest dispersal of the index values will occur when the number of hash keys and the size of the table are prime numbers. Hash functions are one way in that a key can be converted into an index, but not the other way around. Hashing methods are evaluated in Chapter Seven.

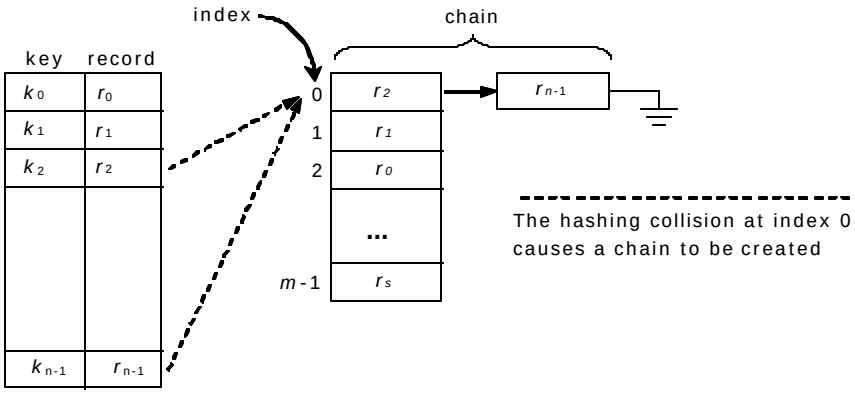

Figure 2.14 – Chained hashing resolution of collision in Figure 2.13

Collisions can be handled inside or outside of the hash table. One method of outside handling is *Simple chain hashing*. A hash table can be implemented as a one dimensional array of pointers to linked lists, and thus overcome the limitation of the fixed size of the hash table, while consuming storage space as needed. When trying to insert a key into a position that already holds another key (collision), the new key can be added to the linked list of keys. Colliding keys are placed in a linked list pointing out from the index location where the keys collided. When a (colliding) key is accessed, the nodes in the list are traversed, starting with the head and advancing towards the tail of the list. Look up time is then proportional to the number of nodes traversed until the search for key is found.

2.8.4 Self-organizing hashing methods

Self-organizing methods are open hash table methods with chained collision resolution that aim to reduce the average key access time by reordering the keys in each resolution chain. If a self-organizing method is used a key is repositioned after it is accessed so that prior knowledge of key accesses is used to reposition most frequently accessed keys closer to the list head. There are several known methods for reorganizing such colliding hash keys. In this dissertation the queueing performance of hash methods is evaluated employing two of them: the *move-to-front* and *transposition* rules [37].

2.8.4.1 H1 hashing

H1 hashing [51] is a method that rearranges the hash table based on the transposition rule. When a key is requested its address in the hash table is found and swapped with the address of the nearby key located next to it, one position closer to the head of the chain.

As a result the chain gets rearranged, but the same hash function could be used to produce addresses on future accesses. If a single entry chain is accessed, no swap is done. H1 hashing is described in more detail in Chapter Five, where its corresponding algorithm is presented together with a new self-adjusting hashing method, called Aggressive hashing.

2.8.4.2 The performance of self-organizing hashes

Colliding keys are placed in a linked list pointing out from the index location where the keys collided. When a (colliding) key is accessed the nodes in the list are traversed one at a time, starting with the head and advancing towards the tail of the list. It is easy to see that look up time is then proportional to the number of nodes traversed until the searched for key is found.

It has been shown that a static hashing scheme yields longer average key access times than a self-organizing scheme [37]. Let keys k_1, k_2,..., k_N are located $1,2,...,N$ positions away from the list head and N is the cardinality on the set of keys. Then the optimal average time to do successful search for the static scheme will be

$$\overline{C}_N = 1p_1 + 2p_2 + ... + Np_N \,, \tag{2.2}$$

where key accesses are independent, with the probability of accesses for key k_i being p_i. If a self-organizing scheme based on the move-to-front rule is used, the average number of comparisons needed to find an item will be

$$\tilde{C}_N = 1 + 2 \sum_{1 \le i < j \le N} \frac{p_i p_j}{p_i + p_j}. \tag{2.3}$$

It has been shown that $\tilde{C}_N$ is less than $\frac{p}{2}$ times the optimal value for $\overline{C}_N$ [12].

It has been shown that given any probability distribution (on the set of key access probabilities) and independent key accesses, an optimal reorganization scheme should employ the transposition rule [56]. In this same study chain reorganization instances are viewed as permutations on the link list nodes. Given an optimal set of such permutations exist, it is shown that the transposition rule provides the optimal number of comparisons. It is not shown however that an optimal set of request sequences (seen as permutations) exists.

Two other studies evaluated the performance of transposition and move-to-front based on amortization heuristics rather than a probabilistic proof [7],[68]. An amortization heuristic takes into consideration the worst case access times for any key access sequence rather than looking at the probabilities of accesses to keys. Rather than looking at worst case number of comparisons for a single access (which is order of list length) the worst case is considered over a sequence of accesses to several keys in turn, where the assumption that key accesses are independent is dropped. The obtained results show that when the independence assumption is dropped, move-to-front performs better than transposition even though probabilistic analyses imposing this assumption lead to concluding that move-to-front has better average search time performance. Some empirical studies have been performed to compare reorganization methods. To the best knowledge no studies have been found which examine *uniformly distributed* keys. Empirical evaluation has already been done on keys extracted from natural language text files, which have been noted to have Zipf's distribution [37]. In this dissertation, the

performance of keys that are uniformly distributed – CRC32 codes – is evaluated. In addition, hash chaining methods that use CRC32 codes as keys to build routing tables for CDNs are studied.

Chapter Three

Scaling from Local Clusters to Distributed CDNs

Building distributed content is different from implementing local server clusters. Scaling URL switching from a local cluster to a distributed cluster adds several new complexities. Server selection methods in the implementation of globally distributed web sites offers difficulties not present in local server clusters due to bandwidth and delay constraints in the Internet. This chapter describes local server clusters and problems with scaling up existing clustering methods for geographically distributed servers.

3.1 Geographically distributed server clusters

The use of globally distributed Web sites for mirroring of popular content is commonly done. For example, apache.org maintains over 180 mirror sites around the world [2]. Such globally distributed server systems can reduce user response times and decrease Internet traffic load. URL routers can be used to automatically redirect requests to individual servers or server sites. URL routing makes it possible for a single site to consist of locally or globally distributed servers. URL routers should send requests to the "best" server. The best server is the one that can give a satisfactory response time to the user and minimize network traffic. Server selection criteria need to include server load, server contents, and network path characteristics.

Web caching is beneficial in improving response time to client and reducing network traffic. To take advantage of caching when selecting the best server, content knowledge has to be available. URL routers need information about both server load and content.

In geographically distributed server clusters, maintaining current load information is difficult given the constraints of the Internet connectivity between individual servers or sites. In local server clusters, the least loaded server is often selected, because load information can be kept current. When selecting the least loaded server in distributed server clusters server performance could be very poor, because they rely on delayed load information. This poor performance is due to the so-called "herd effect" (which has been studied in [15] and [43]) where requests rush to the currently advertised least-loaded server and cause unbalanced and oscillating loads. Dealing effectively with delayed load information is an open challenge. This dissertation does not address the herd effect problem. This dissertation addresses the effects of delay and bandwidth constraints of Internet connections between distributed sites.

3.2 Building distributed content

Web content is "globalized" via distributed server and caching infrastructures. In a globally distributed web site, comprised of multiple local sites, the local sites partially or fully mirror their contents. Each local site contains one or more servers (of possibly different capabilities) and a reverse cache. At any given time the load and reverse cache contents vary between the sites. Each local site is front-ended by an application level switch, called a URL switch. URL switches are described in detail in Chapter Four.

These switches have knowledge of state information such as site load and contents and use this information to make routing decisions.

3.2.1 Implementing local server clusters

The servers and caches that form a local server cluster site are connected within a high-speed switched LAN. Such local server clusters can be implemented with a front-end application layer switch. The simplest and most common policy for server selection is for the switch to direct a request to the least loaded server in the local site. The high-speed LAN enables state information to be frequently shared and least-loaded server selection to be effective. If the URL servers "spoofs" for the multiple servers of the local site, Network Address Translation (NAT) methods can be used to switch requests to servers. TCP connection handoff and splicing are other methods. Content-based server selection in the context of a local server cluster has been investigated [50]. The front end switch distributes requests to servers in a local cluster based on a Locality Aware Request Distribution (LARD) technique which combines load balancing and content knowledge to achieve both load balancing and increased cache hits at back-end servers. LARD is a technique that assigns documents to servers. Incoming requests are dispatched based on knowledge of the content at each server. Each incoming request is satisfied by some server that contains the requested document. This server is also selected to be less loaded at the time of the request. The data structures used in this study for identifying servers with cached objects are unclear. A load factor is used to determine when to favor an "uncached" server over a server with the requested object in cache. Local clusters can be implemented without front-end switches. In Distributed Packet Rewriting (DPR) [6] the

individual servers in a cluster are let to route connections to less loaded servers. Thus, DPR implements routing functions within a server cluster. The most effective metric for server load was found to be number of TCP connections [6].

3.2.2 Implementing distributed server clusters

Scaling URL switching from a local cluster to a distributed cluster adds several new complexities:

1. Internet connections that connect two different local sites have less bandwidth and greater delay than a the connections within the local LAN. Thus, state information must be shared with less frequency and be smaller in size.
2. Network path characteristics between a client and a server site must be considered if a request is switched to a distant site.
3. Switching techniques that work in a local site would result in a "dog leg" routing between two distant sites (that is, DPR and NAT). Thus, redirection techniques need to be explored.

In addition, the different embedded components of a single web page – text, image, etc. may be downloaded from two or more remote sources. Hyperlinks that appear on some web page are not a part of the that page, that is, they are not embedded into the HTML document and are not considered embedded components. It has been shown through experiments that downloading the different components of a single web page from multiple servers results in worse than optimal performance [34]. Client delay is reduced by 30% when servers are switched for the download of document components, as compared to 90% reduction when a nearby-distributed server satisfies the request. Hence,

throughout this dissertation is assumed that all components are downloaded from the same source.

3.3 Adding content knowledge to server selection

URL switches must be able to exploit knowledge of both server load and content. Content information should be shared between URL switches. The information consists of a data structure that contains <server, content> tuples. A key issue is cache validity time – how long a previously requested object will remain in a site cache. The digests will need to be updated on a frequency approximating cache validity times. Thus, an updated digest guess can be used to adjust the known load of the server. Both the server load value and partial knowledge of cache contents are estimates – the load due to its delay in reporting and the cache contents due to both the probabilistic nature of digests and the possibility that an object previously known to be in cache has been since removed from cache.

A combined load and content metric has been suggested for use for selecting the best server in a globally distributed site [23]:

$$X_j = \frac{(L_{total} - (L_j - a_j))}{L_{total}} \tag{3.1}$$

The related method for server selection is based on load information (L_j) that uses content knowledge to adjust the calculation of the server selection (X_j) term. L_{total} is the sum of the queue lengths of uniformly randomly chosen K number of servers, and a_j is an adjustment factor that is positive when it is assumed that server j may contain the requested object in its cache. The a_j value is updated periodically. The value of a_j is a

function of how much serving a cached object requires less server load than serving a disk- resident object. For example, a_j can be the number of connections that a cache can independently handle (and thus relieving the server of this load).

Chapter Four

System Design of a URL Router

This chapter describes an application-layer router that is capable of intercepting HTTP requests and redirecting them to the best source from which to be satisfied. Such a router is known as a *URL router* [66]. A URL router makes forwarding decisions based on the URL contained in an HTTP request. URL routers are application layer routers [3] (also called layer-7 switches based on the ISO layered model). A URL router can front-end an origin site that contains either one server, or a cluster of servers; a caching infrastructure; or it can serve as a proxy device on the client side. Figure 4.1 shows the possible locations of a URL router relative to caches and servers. In this dissertation, the focus is on HTTP redirection, but the methods developed apply equally to DNS redirection and TCP connection spoofing/splicing. A server sending an HTTP redirection response message to a client can directly use the HTTP 302 response. The HTTP 302 redirection message instructs the client browser to automatically re-request the object at another host ID. HTTP redirection requires a double network round-trip delay, which is significant for small requested objects but not for larger objects. HTTP redirection also requires significant server resources, since the request must be processed through all layers of the protocol stack and handled directly in software. Generating an HTTP redirection response requires the same workload (on a possibly overloaded server) as serving the requested content. This further motivates the need for URL routers to front-end content sources.

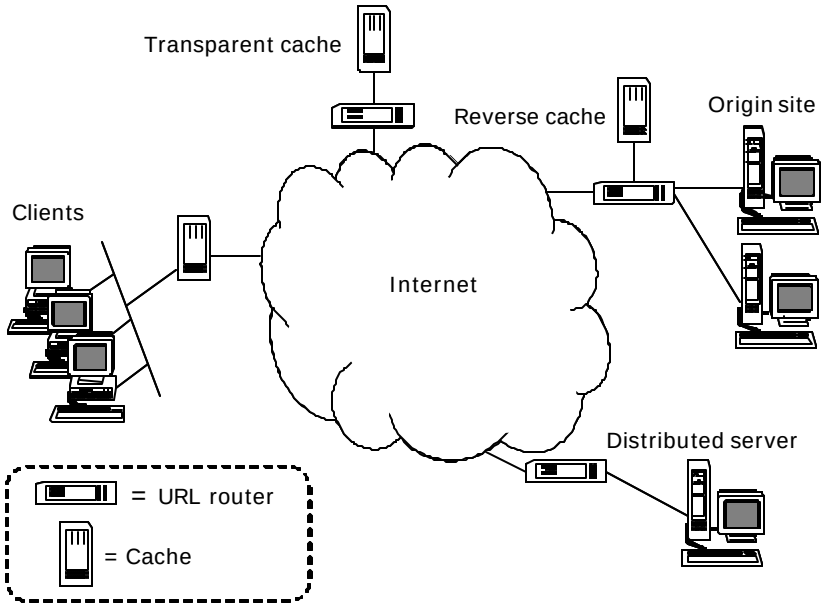

Figure 4.1 – Placement of URL routers in the Internet

4.1 Operation of a URL Router

The semantics of an HTTP request require that a TCP connection first be established before the request is sent. A URL router must have an IP address that is either the proxy address, or that represents the content source. Following connection establishment, the URL router can return an HTTP redirection message to the client and terminate the connection. HTTP redirection automatically forces the client to resend a request to a new content source specified in the HTTP redirection message.

The first generation URL routers [35] were used to front-end content sources and forward incoming client requests to the best server in a local server cluster. The best server could be determined by dynamic metrics such as server load or static metrics such as matching the server type to the requested content type (that is, forwarding requests for

streaming video to a specialized video server). TCP connection spoofing or splicing was used to maintain the client connection established to the URL router. The second generation URL routers added the ability to redirect requests to geographically distant content sources using HTTP redirection mechanisms. If all servers in a cluster were above a threshold in number of connections, the URL router would send an HTTP redirection message to the requesting client. This HTTP redirection forces the client to re-request the content from a predefined "overflow" server. Figure 4.2 shows the flow of a request being forwarded or redirected (DNS redirection is not shown here). A request is forwarded locally, within the origin server cluster (1), or an HTTP 30X response code is sent back to the client, forcing the client to automatically re-request the content from a distributed server (2).

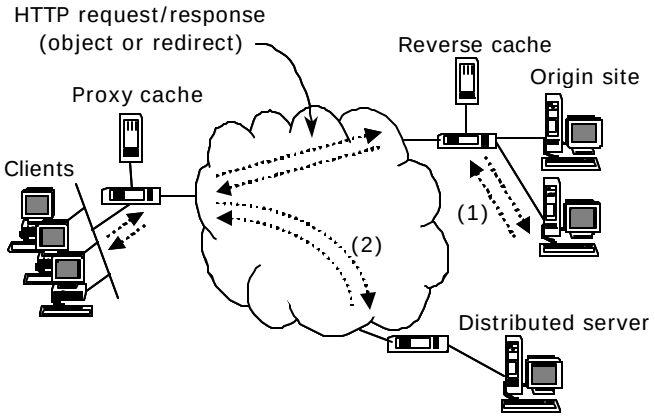

Figure 4.2 – URL router forwarding or redirecting a request

A URL router does the following:

1. Establishes a TCP connection with a client (initiated by the client).
2. Receives and parses an HTTP request from a client.

3. Looks-up the requested URL in a routing table and determines the IP address of the best content source.
4. Spoofs or splices the connection with the content source if the content source is local to the URL router.
5. Sends the client a redirection message containing the new URL of the content source if the content source is remote to the URL router.

The URL router must maintain a *URL routing table* from updates received from other URL routers or directly from content sources. The URL routing table is built from *URL lists* where a URL list contains the URL and the host IP addresses which contain the URL. URL routing tables are described later in this paper.

4.2 Architecture of a URL Router

A URL router can be implemented with a general purpose computer attached to a layer-3 switch (that is, as a "one armed" router as shown in Figure 4.3), or with specialized hardware within a layer 3 switch. In the latter case, the URL router is its own specialized device. A specialized URL router is similar in design to an IP router. The URL router consists of a switch core with attached line cards. The URL routing table can be maintained in a global memory and transferred fully or partially to local memory on the line cards. Switching decisions are made by the line cards whenever possible. For a URL router implemented within a PC as a one armed router, the layer 3 switch forwards to the URL router TCP connection requests (that is, SYN packets) designated for the IP address of the represented content source.

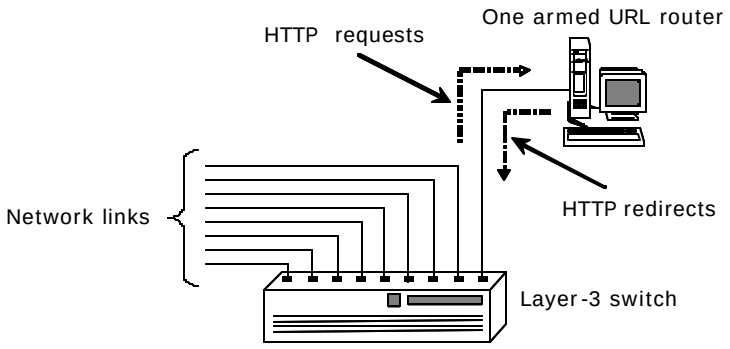

Figure 4.3 – A one-armed URL router

4.3 Structure of a URL routing table

A URL routing table is used to determine where to redirect a request. The table contains URLs and the locations of content sources. Associated with every content source can be the load and distance state. This state information, along with knowledge of the client's location, is used to select the best content source. Figure 4.4 shows the structure of a routing table with N URLs and M_i $(i = 1,2,\ldots,N)$ locations per URL. The content stored at a source varies over time and thus updates must regularly be shared between distributed URL routers. It is desirable to reduce the overhead required to send the URL lists and to reduce the processing complexity (and thus the time) needed to perform look-ups.

4.4 Performance bottlenecks for a URL router

For a one armed URL router based on a standard PC or workstation, a single CPU handles all the tasks. Thus, a single CPU needs to handle both TCP connections and URL look-ups (as well as routing table updates). Memory is a constraint if the URL

routing table cannot fit entirely into main memory. If the routing table must be swapped out of disk storage, look-up times will be greatly increased. For a specialized URL router, a single-line card will likely contain two CPUs (or one CPU and one special-purpose hardware circuit). One CPU is for handling TCP connections, and the second CPU (or ASIC) is for URL look-up. Memory is a significant constraint on a line card. Thus, performance issues exist with swapping between a centrally stored main URL routing table and the partial tables on the line cards. In the following chapters of this dissertation, the memory constraints and CPU bottlenecks for URL look-up are addressed by using signatures to reduce the length of URLs and enable efficient hashing-based look-up.

URL 1	IPaddr1(state), IPaddr2(state), ... M_1(state)
URL 2	IPaddr1(state), IPaddr2(state), ... M_2(state)
URL 3	IPaddr1(state), IPaddr2(state), ... M_3(state)
...	...
URL *N*	IPaddr1(state), IPaddr2(state), ... M_N(state)

Figure 4.4 – The structure of a URL routing table

Chapter Five

Using Signatures for URL Routing

This dissertation proposes the use of signatures to reduce variable length URL strings into fixed-length integer signature values to be used as keys in the URL routing table. In existing methods [3], the entire URL is stored as the key value. In this chapter, CRC32 signature generation methods are described and their application to signature based digesting for URL routers is presented.

5.1 Using CRC32 for URL signatures and digesting

CRC32 codes are generated for each URL string (obtained in turn from an HTTP trace file) in the routing table. The CRC32 signatures of the original URL strings are used as keys to build the hash table, since CRC32 codes are well distributed, unlike URL strings that contain natural language constructs and words and hence are not uniformly distributed. Then, a subset of the 32 bits representing each signature are used to produce a hash index. If the 24 most significant bits are taken to produce hash indexes, the key (h is hexademical and b is binary) 87654321h, will hash to $h(x) = h(87654321h) = h(10000111011001010100001100100001b) = 100001110110010101000011b$.

Table 5.1 shows some widely used CRC polynomials, including the polynomial for CRC32 used in this dissertation to generate signatures of URL strings. The CRC-12 is used for transmission of streams of 6-bit characters and generates 12-bit frame check sequence. Both CRC16 and CCRCCCITT are used for 8-bit transmission streams and

both result in 16-bit frame check sequences. The last two are widely used in the USA and Europe respectively and give adequate protection for most applications. Applications that need extra protection can make use of the CRC32 which generates a 32-bit frame check sequence. The use of CRC32 is part of the IEEE 802 standard.

Table 5.1 – Commonly used CRC polynomials

CRC12	$P_{12} = x^{12} + x^{11} + x^3 + x^2 + x^1 + x^0$
CRC16	$P_{16} = x^{16} + x^{15} + x^2 + x^0$
CRCCCITT	$P_{CCITT} = x^{16} + x^{12} + x^5 + x^0$
CRC32	$P_{32} = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} +$ $+ x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + x^0$

A good signature algorithm will generate unique signatures for different URLs. A URL list is reduced – one signature generated for each URL – using software or hardware generation of signatures at the content source, and is then sent to the URL routers responsible for routing requests for the stored content. The routing table is built using hash table methods with signatures (and not full-length URLs) as the keys. Thus, the structure of the routing table is that of a hash table. The CRC codes are generated on-the-fly when an HTTP request is received, and are then used in the URL routing table look-up. A hash code for a URL is the first H bits of its CRC32 (for $1 \le H \le 32$) for a hash table of 2^H entries. CRC codes can be generated on the fly with very simple hardware circuits when a packet is received. A CRC32 results in 2^{32} possible signatures. The

hardware required to generate a CRC is a serial shift register and XOR circuit; the hardware is an integral part of all network adapters. At the URL router, the URLs of all incoming HTTP requests are serially encoded into a CRC.

In the case of a one-armed URL router, the network adapter can generate the CRC (or the CRC can be generated by software). Existing network adapters cannot be programmed to generate CRCs on subfields within a received packet; however, it is possible to use the single CRC circuit of a network adapter to simultaneously generate the packet CRC and CRCs on any subfields of the packet if partial CRC values can be accessed. A single CRC32 circuit can be used to simultaneously generate an overall packet CRC32 and a CRC32 code for a subfield (on byte boundaries) of the packet will be described. It is assumed that the current value of the CRC32 shift register can be sampled at byte boundaries.

The CRC properties described in [59] and proved in [32] are used. Let P be a CRC generator polynomial. For any polynomials (bit sequences) A_i, $i = 1, \ldots, m$ there are the following properties:

$$\text{Rem}\left(\frac{\sum_{i=1}^{m} A_i}{P}\right) = \text{Rem}\left(\frac{\sum_{i=1}^{m} R_{A_i}}{P}\right) \tag{5.1}$$

and,

$$\text{Rem}\left(\frac{\prod_{i=1}^{m} A_i}{P}\right) = \text{Rem}\left(\frac{\prod_{i=1}^{m} R_{A_i}}{P}\right) \tag{5.2}$$

where,

$$R_{A_i} = \text{Rem}\left(\frac{A_i}{P}\right). \tag{5.3}$$

Let A_0 be the bit sequence from the beginning of the packet to the last byte before the beginning of the subfield, A_1 be the bit sequence from the beginning of the packet to the last byte of the subfield, and A_2 be the bit sequence of the subfield. Let the subfield A_2 be of length M bits. The remainders corresponding to all possible lengths of A_2, that is, $R_M = \text{Rem}\left(2^M / P\right)$, are stored in a look-up table. $R_{A_0} = \text{Rem}\left(A_0 / P\right)$ and $R_{A_1} = \text{Rem}\left(A_1 / P\right)$ are returned from the CRC32 circuit (that is, from the network adapter). $R_{A_2} = \text{Rem}\left(A_2 / P\right)$ must be found to have the CRC32 for the subfield A_2. R_{A_2} is solved for as follows: Let A_3 be the bit sequence A_0 shifted left M bits (that is, multiplied by 2^M). Getting R_M from a look-up table, from (5.2) the following is obtained:

$$R_{A_3} = \text{Rem}\left(\frac{A_0 \cdot 2^M}{P}\right) = \text{Rem}\left(\frac{R_{A_0} \cdot R_M}{P}\right) \tag{5.4}$$

Now a subtraction can finally be performed, since A_3 is of same length as A_1. From Equation (5.1) the following is obtained:

$$R_{A_2} = \text{Rem}\left(\frac{A_3 - A_1}{P}\right) = \text{Rem}\left(\frac{R_{A_3} - R_{A_1}}{P}\right) \tag{5.5}$$

Equation (5.5) follows from modulo-2, addition and subtraction being the same (that is, an XOR operation). Solving for Equation (5.4) requires a 32-bit multiplication. A 32-bit multiplication can be implemented in software on a 32-bit processor using the algorithm of Figure 5.1.

```
; Inputs are m1 and n1, output is r2:r1 = m1 * n1
; where r2 is the high-order 32-bits of the
; product.  All integers are assumed to be 32-bit
; and unsigned.
PROCEDURE MULT64(m1, n1, r2, r1)
BEGIN
  INTEGER m1, n2, n1, r2, r1
  INTEGER count
  ; Initialize to zero
  n2 = r2 = r1 = 0
  ; Multiply loop
  LOOP count = 1 TO 32
    IF (m1 AND 1) THEN
      r2 = r2 XOR n2
      r1 = r1 XOR n1
    SHIFTLEFT(n2)
    IF (n1 AND 80000000H) THEN
      n2 = n2 OR 1
    SHIFTLEFT(n1)
    SHIFTRIGHT(m1)
END
```

Figure 5.1 – 32-bit multiply algorithm for 64-bit product (from [59])

5.1.1 Handling of Collisions

A false hit or "collision" will occur when the CRC signatures of different URLs are the same. This may result in an HTTP request being redirected (mis-routed) to a content source that does not contain the requested object. In the case of redirection to a cache, the cache will request the object (often from the origin site) and store it for future redirections. In the case of redirection to a distributed server, the mis-routing is handled one of two ways. One method is to prevent collisions by checking CRCs during file generating and forcing a file renaming if a collision occurs. The second method is to use

HTTP redirection to force another redirection to the origin site; this double re-direction will occur only very rarely.

For 2^K unique hashes (K = 32 for CRC32) and N URLs,

$$\Pr[\text{collision in a set of } N \text{ URLs}] = 1 - \prod_{i=1}^{N} \frac{\left(2^K - i + 1\right)}{2^K} \tag{5.6}$$

For a given URL in a list of N URLs,

$$\Pr[\text{collision for a given URL}] = 1 - \left(\frac{2^K - 1}{2^K}\right)^{N-1} \tag{5.7}$$

The expected (mean) number of collisions in a set of N URLs is then,

$$\text{E[number of collisions]} = N\left(1 - \left(\frac{2^K - 1}{2^K}\right)^{N-1}\right) \tag{5.8}$$

5.2 Self-adjusting hashing method for fast routing table look-ups

Hash functions should have low collision rates. That is, it should be difficult to make strings that produce the same key. A hash table can be implemented as an array of pointers to linked lists, and thus the limitation about the fixed size of the hash table can be eliminated, while consuming storage space as needed. When trying to insert a key into a position that already holds another key (this is a collision), the new key can be simply added to the linked list of keys. When a (colliding) key is accessed the nodes in the list are traversed one at a time, starting with the head and advancing towards the tail of the list. Look up time is proportional to the number of nodes traversed until the search for key is found. The routing table of a URL router can be a hash table of URL strings as

keys and IP addresses (of content servers) as records. By using URL signatures instead of full URLs as keys, the size of the routing table is reduced. This dissertation investigates the use of URL signatures for both building reduced-size routing tables and as hash indexes. It is expected that requests for web content will have very high temporal locality. This suggests that self-organizing hashes that can put popularly requested URLs at the head of a hash chain may be well suited for improving look-up time.

A second key contribution of this dissertation is a new self-organizing hashing method called Aggressive hashing based on the move-to-front rule used for self-organizing a chained hash table. In Aggressive hashing, the currently accessed record is moved to the head of its chain. The average worst case look-up time for the entire table is $N/|hash\_indexes| = N/2^K$ (or the mean chain length), where N is the number of keys.

Figure 5.2 shows the Simple chained hashing (presented in Chapter Two) search and insertion algorithm based on the hashing algorithm from [37]. Each resolution chain is denoted by a one-dimensional array LIST*i*, where *i* is the hash table index. A key to be looked-up or inserted is denoted by KEY and TEMP is a temporary value holder used to perform a swap operation. Step C4 from the Simple chain hashing algorithm is modified to step C4B for Aggressive hashing. Note that the H1 algorithm, described in Chapter Two is based on a similar replacement of step C4 with C4A algorithm. The chain shift described by the for loop in step C4B can be implemented by two simple pointer reassignment statements; this has negligible time complexity.

C1. [*Create lists.*] **For** $i \leftarrow 0$ to $m-1$ set LIST$i \leftarrow$ NULL.
C2. [*Hash.*] Set $i \leftarrow h$(KEY), $j \leftarrow 0$ (Now $0 \leq i \leq m-1$.)
C3. [*Is there a list?*] If LISTi = NULL, go to C6. (Otherwise LISTi is occupied; will look at the list of occupied nodes that starts here.)
C4. [*Compare.*] **If** K = LIST$i[j]$, the algorithm terminates successfully.
C5. [*Advance to next.*] **If** LIST$i[j] \neq$ NULL, set $j \leftarrow j+1$ and **go to** step C4.
C6. [*Insert new key.*] Set LIST$i[j] \leftarrow$ KEY.

C4A: [*Compare and transpose – H1 hashing*]
If K = LIST$i[j]$ and $j \neq 0$, swap LIST$i[j]$ with LIST$i[j-1]$ and terminate algorithm with success.
Else terminate algorithm with success.

C4B: [*Compare and move-to-front – Aggressive hashing*]
If K = LIST$i[j]$ and $j \neq 0$ LIST$i[j] \leftarrow$TEMP, for k = 0 to j LIST$i[k] \leftarrow$ LIST$i[k-1]$. Terminate algorithm with success.
Else terminate algorithm with success.

Figure 5.2 – Hashing algorithms (based on [37])

Chapter Six

Evaluation of URL Signatures for Digesting

In this chapter, the use of CRC32 for building URL digests is evaluated. The evaluation measures CPU time and memory size of CRC32 and other digesting methods from the literature. Methods of reducing the size of URL lists, resulting in reduction of network bandwidth to share the URL lists, are of interest. These reduced URL lists are called *digests*, and the methods for sharing digests among routers are called *digesting*.

6.1 Access lists used in the evaluation

Representative URL access logs were used to evaluate the performance of digesting methods. A *URL list* was generated by taking the unique values from the access list. A digest was generated for each URL list using CRC32 signatures for each URL. The process of generating a digest is as follows. Figure 6.1 shows a snippet from a raw HTTP access log (due to line length limits, the lines are cut and marked with a "..."). This snippet contains duplicate URLs and one URL for dynamic content. The relevant field of interest in the raw HTTP access log is the URL. The URL access list, created from the HTTP access log is a list of URLs as shown in Figure 6.2. The URLs in the HTTP access log associated with dynamic content were discarded. URLs with the "cgi" substring were removed. This is because dynamic content must always be obtained from an origin server (dynamic content cannot be cached). From the URL access list, a URL list is generated. A URL list is the unique set of URLs from a URL access list. The Unix

`uniq` command was used to extract the unique set of URLs. Figure 6.3 shows a URL list. Thus, a URL list is a key-value list and an URL access list is a look-up list. The URL lists were used in two formats. The first format is with URL strings as keys as in Figure 6.3. The second format is with URL strings reduced to CRC32 signatures, shown in Figure 6.4.

```
30633 73.139.209.145 TCP_MISS/200 74052 GET http://gigex1.com/M0016 ...
14104 227.229.152.199 TCP_MISS/200 17048 GET http://fourohfour.xoom ...
13549 73.139.209.145 TCP_CLIENT_REFRESH_MISS/200 368 GET http://www ...
704 148.97.138.187 TCP_REFRESH_MISS/504 1339 GET http://www.rocksho ...
706 112.211.98.27 TCP_DENIED/403 1149 GET http://store2.yimg.com/I/ ...
707 244.60.215.3 TCP_MISS/503 1265 GET http://www.linkexchange.ru/c ...
33990 227.229.152.199 TCP_MISS/200 4192 GET http://fourohfour.xoom. ...
30461 227.229.152.199 TCP_MISS/200 5652 GET http://fourohfour.xoom. ...
768 112.211.98.27 TCP_DENIED/403 1133 GET http://www.body-n-mind.co ...
176 112.211.98.27 TCP_DENIED/403 1149 GET http://store2.yimg.com/I/ ...
151 112.211.98.27 TCP_DENIED/403 1133 GET http://www.body-n-mind.co ...
119622 163.197.198.77 TCP_MISS/200 21880 GET http://detik.com/peris ...
79 112.211.98.27 TCP_DENIED/403 1133 GET http://www.body-n-mind.com ...
239 186.19.182.251 TCP_MISS/200 265 POST http://srv01.lingocom.com/ ...
```

Figure 6.1 – Snippet of a “raw” HTTP access log

```
http://gigex1.com/M0016500027/00000/MDK2.exe
http://fourohfour.xoom.com/Members404Error.xihtml
http://www.americangreetings.lycos.com/images/sidedoor/space.gif
http://www.rockshox.com/media/images/images02/001_r08_c37.gif
http://store2.yimg.com/I/greatsword_1565_161175
http://fourohfour.xoom.com/Members404Error.xihtml
http://fourohfour.xoom.com/Members404Error.xihtml
http://www.body-n-mind.com/haunted3.jpg
http://store2.yimg.com/I/greatsword_1565_301888
http://www.body-n-mind.com/haunted3.jpg
http://detik.com/peristiwa/2000/08/01/200081
http://www.body-n-mind.com/italian4.jpg
http://srv01.lingocom.com/scripts/engine
```

Figure 6.2 – Snippet of an URL access list created from the log snippet of Figure 6.1

```
http://gigex1.com/M0016500027/00000/MDK2.exe
http://fourohfour.xoom.com/Members404Error.xihtml
http://www.americangreetings.lycos.com/images/sidedoor/space.gif
http://www.rockshox.com/media/images/images02/001_r08_c37.gif
http://store2.yimg.com/I/greatsword_1565_161175
http://www.body-n-mind.com/haunted3.jpg
http://store2.yimg.com/I/greatsword_1565_301888
http://detik.com/peristiwa/2000/08/01/200081
http://www.body-n-mind.com/italian4.jpg
http://srv01.lingocom.com/scripts/engine
```

Figure 6.3 – Snippet of an URL list from an HTTP access list showing full URLs

```
3B39227Eh
2064CE70h
AF724180h
89B96C0Ah
8BD5FCB1h
EA6898CCh
7CB781B4h
DFA3E708h
4E56EC0Eh
10EB737Bh
```

Figure 6.4 – The URL list of Figure 6.3 with CRC32 URL signatures

6.2 Evaluation of CRC32 digesting

The use of URL signatures for URL routing was evaluated in terms of three criteria. The three criteria, and thus also the response variables for the experiments, were:

1. Probability of false hits due to signature collisions
2. Processing (CPU) resources required to generate URL signatures at the content sources
3. Reduction in processing and memory resources for URL look-up by signature versus full URL

From each access list, a URL list was generated by taking the unique values from the access list. The URL list is the list of objects stored in a cache or server. For each URL

list, a compressed form was generated using CRC32 signatures for each URL. For look-up performance measurements, the URL list was stored as a Simple chained hash table with either:

- Full URL as the key
- Signature (CRC32) of the URL as the key

For each key, a four-byte value was stored representing an IP address of a content source in a URL routing table. The URL list hash table is of length 2^H entries (H = number of bits in the hash value taken as the first H bits of the URL CRC32). For the hash table with full URLs as keys, the hash values were taken as the first H bits of the URL CRC32. The value of H was a control variable in the performance measurement. A larger H results in greater memory usage, but smaller hash chains will give faster look-ups. Table 6.1 summarizes the access logs used in this evaluation. Table 6.1 shows the mean URL length in each access list and the size of the URL list with full URLs and with CRC32 signatures. The CRC32 digested URL list size is smaller than the full URL list size by a factor of the mean URL length divided by four (for four bytes in a CRC32).

Experiment #1: Evaluation of the number of false hits or CRC collisions in the access and URL lists was measured. A CRC32 was generated for each URL and the number of non-unique CRC32s counted.

Experiment #2: Evaluation of the CPU time required to generate the compressed URL list based on CRC32 with software CRC generation. An 8-bit look-up method of generating CRCs was implemented in C language (using the methods described in [65]).

Table 6.1 – Summary statistics for access logs used in this evaluation

Access list name	Number of accesses	Number of URLs	Mean URL len (bytes)	Full URL list size (bytes)	CRC32 list size (bytes)
www.peak.org [52]	16,374	70	23.93	1,675	280
SDMA [61]	41,941	153	33.76	5,165	612
UVA [71]	318,899	45,816	44.91	2,057,625	183,264
NLANR [49]	944,028	504,967	58.44	29,510,135	2,019,868
UC Berkeley [46]	1,791,349	149,344	41.87	6,253,716	597,376
mcs.net [40]	1,862,070	75,361	29.87	2,250,829	301,444
Hyperreal.org [5]	4,080,590	86,338	89.17	7,698,337	345,352
CA*netII [63]	4,642,861	2,552,045	57.83	147,573,556	10,208,184
USF CSEE [16]	8,819,454	49,029	51.84	2,541,483	196,116

Experiment #3: Evaluation of the CPU time required to look-up all the entries from an access list in the associated URL list was measured. The URL list was stored as a Simple chained hash table with either a full URL as the key or a CRC32 URL signature as the key. To study the effect of hash table size, the number of entries was varied from 1024 (H=10) to 4,194,304 (H=22).

6.2.1 Experiment results

Table 6.2 shows the results for experiment #1 in terms of collisions for the access and URL lists. The percentage of collisions grows with the size of the URL list. There is no noticeable pattern in terms of access list collisions. Table 6.2 shows URL list collision probabilities and the expected number of collisions from Equation 5.8. CRC32 signatures

Table 6.2 – CRC32 URL list collision probability

Access list name	Collisions Measured	Calculated value	Pr[collision] measured
www.peak.org	0	0	0.0000000
SDMA	0	0	0.0000000
UVA	0	1	0.0000000
NLANR	68	59	0.0001347
UC Berkeley	2	5	0.0000134
mcs.net	0	1	0.0000000
hyperreal.org	2	2	0.0000463
CA*netII	1558	1516	0.0006105
USF CSEE	2	1	0.0000408

of URLs appear to be uniformly random with the measured and expected number of collisions close to the same.

Table 6.3 shows the results for experiment #2 with CPU measurements accurate to a precision of 10 milliseconds. Time to generate the list and average time to generate a URL signature is shown. These results show that the compression time is usually a

Table 6.3 – CPU time to generate a compressed URL list

Access list name	Time for URL list	Time for URL
www.peak.org	<10 millisec	--
SDMA	<10	--
UVA	40	0.8730 microsec
NLANR	460	0.9109
UC Berkeley	100	0.6695
mcs.net	40	0.5307
hyperreal.org	120	1.3897
CA*netII	2390	0.9368
USF CSEE	40	0.8158

fraction of a second, with each URL requiring about 1 microsecond to be compressed to a CRC32 signature.

Figure 6.5 shows the results for experiment #3 as the total time to look-up 50,000 entries from a URL list based on the NLANR log [49]. As *H* decreases, the mean chain length in the hash table increases causing a greater number of memory references to be required to match a key. For *H*=12, using URL signatures results in 6 times reduction in look-up time. Beyond about *H*=14, increasing the hash table size has little additional benefit to reducing look-up time. At *H*=14, the rate of look-ups is about 2.9 million per second for compressed URLs and about 440 thousand per second for full URLs (with 100% CPU utilization in both cases).

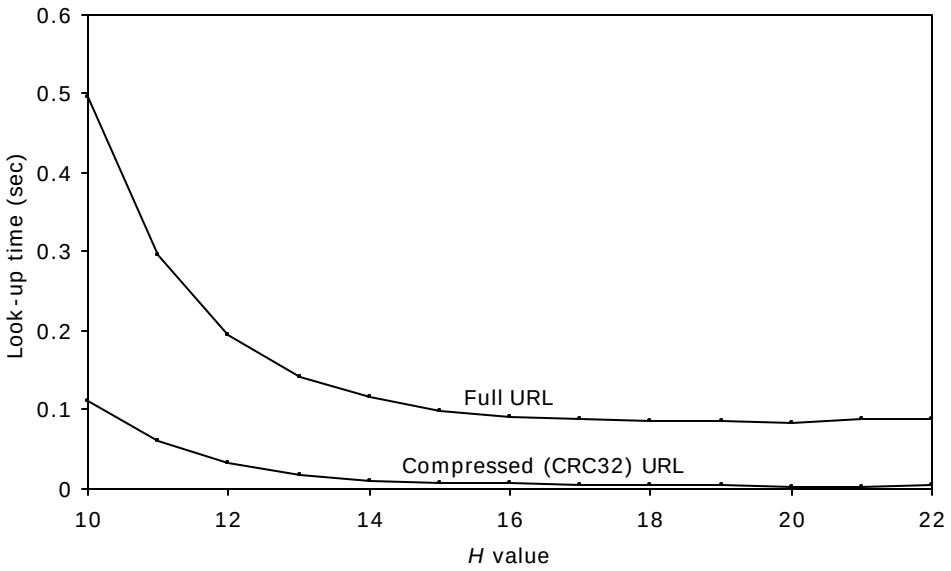

Figure 6.5 – The effects of hash table size on look-up time

6.3 Comparing CRC32 to MD5-Bloom filter digesting

CRC32 digesting was compared to existing Bloom filter digesting. Trace-driven simulation was used to evaluate URL digesting methods. The input was the URL list that was to be compressed into a digest. Four URL access logs were used to derive the URL lists. The access logs were: CA*net squid proxy log (CA *net list) [63], a log from the USF Computer Science and Engineering server (CSE list), an NLANR proxy log (NLANR list) [49], and a server log from Virginia Tech (VTech list) [71]. URL lists #3 and #4 are the same as used in Summary Cache [19]. Table 6.4 summarizes the URL lists. The last column is the mean number of parts in a URL, which is used in experiment #4 (URL parts are substrings of the URL divided by each of the forward slashes "/" in it. For example http://www.some.com/folder1/file.name has the string "folder1" as one of its parts).

Table 6.4 – Summary statistics for the URL lists used in this evaluation

	# URLs	Size	URL len	URL parts
CA*net list	2,552,045	140.75 MB	56.8 B	6
CSE list	49,029	1.54	28.8	7
NLANR list	483,631	16.30	56.0	6
VTech list	45,817	1.96	43.9	6

Experiment #1: Evaluation of digest size and CPU time. For the MD5-Bloom filter, CRC32, 32-bit checksum, and Lempel-Ziv (LZ) compression (pkzip25 [53] was used for LZ compression). For the MD5-Bloom filter digests, a load factor of 8 was used.

Experiment #2: Evaluation of MD5-Bloom by varying load factor. Again, digest size and CPU time are measured.

Experiment #3: Evaluation of the effects of URL length on collisions. A comparison of MD5-Bloom and CRC32 for collision rate as a function of URL length. Only URLs of greater than 25, 30, …, 80 bytes and load factor 8 were used.

Experiment #4: Evaluation of digest size of the hash chain method [41]. The digest size is based on the number of components in a URL and a shared tree structure with 32 bits to represent a <depth, hash code> pair.

6.3.1 Experiment results

Tables 6.5 and 6.6 show the results for experiments #1 and #2 for the four URL lists. The last three rows show the results for MD5-Bloom digesting with varying load factor (experiment #2). The CRC32 digest requires about 6 times less CPU time and with no increase in collisions! LZ compression requires less CPU time than Bloom filter and slightly more time than the CRC32 method, but results in larger digest sizes. The 32-bit checksum yields only slightly better CPU time results than CRC32 at an expense of higher collision rates.

For experiment #4, the digests of the four URL lists were larger than all other methods from experiment #1 with the exceptions of MD5-Bloom filter with load factor 32 for all URL lists and LZ and MD5-Bloom with load factor 16 for URL CSE list. Hash chaining produces an average of 212% larger digests than CRC32. CRC computation in

software is linear with the CRC length, hence the hash chain method will require greater CPU time than the CRC32 method.

Table 6.5 – Results for experiment #1 and #2 for CA*net and CSE lists

	CA*net list			CSE list		
Method (L. Factor)	CPU time (sec)	Size (Mbytes)	Collisions (%)	CPU time (sec)	Size (Mbytes)	Collisions (%)
MD5-Bloom (8)	89.13	9.74	0.03	1.63	0.19	0.00
CRC32	16.22	9.74	0.03	0.27	0.19	0.00
32-bit checksum	14.85	9.74	0.71	0.24	0.19	0.22
LZ compression	17.35	16.43	0.00	0.23	0.25	0.00
MD5-Bloom (8)	89.13	9.74	0.03	1.63	0.19	0.00
MD5-Bloom (16)	92.37	19.47	0.00	1.71	0.37	0.00
MD5-Bloom (32)	97.40	38.94	0.00	1.84	0.75	0.00

Table 6.6 – Results for experiment #1 and #2 for NLANR and VTech lists

	NLANR list			VTech list		
Method (L. Factor)	CPU time (sec)	Size (Mbytes)	Collisions (%)	CPU time (sec)	Size (Mbytes)	Collisions (%)
MD5-Bloom (8)	16.25	1.84	0.01	1.52	0.17	0.00
CRC32	2.51	1.84	0.01	0.20	0.17	0.00
32-bit checksum	2.29	1.84	0.30	0.20	0.17	0.08
LZ compression	3.32	4.00	0.00	0.27	0.31	0.00
MD5-Bloom(8)	16.25	1.84	0.01	1.52	0.17	0.00
MD5-Bloom (16)	16.99	3.69	0.00	1.59	0.35	0.00
MD5-Bloom (32)	18.20	7.38	0.00	1.70	0.70	0.00

Figure 6.6 shows experiment #3 results for CA*net list, the other URL lists exhibit a similar trend. As the URL length increases, the collision rate decreases.

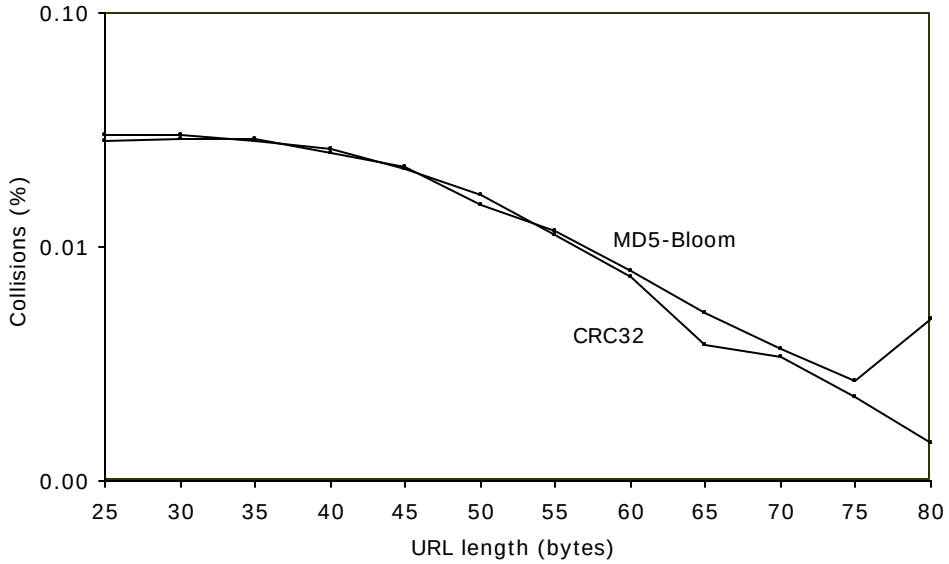

Figure 6.6 – Results from experiment #3

6.4 Discussion of results

URL signatures can be used to improve URL routing. URL signatures based on CRC32 can reduce by a factor of 15 the size of URL lists. URL signatures can also speed-up the look-up of URLs in a hash table by a factor of 7. A reduced URL list requires less network bandwidth to transfer and less memory for storage in the URL router. For CRC32, the percentage of collisions grows with the size of the URL list, as expected, however the number of collisions in a typical access list was found to be slightly better than the number of collisions resulting from using existing methods (0.74% smaller). Compared to existing URL digesting implementations, the CRC32 digest was shown to

require less CPU resources to generate the digest at a content source and produce equivalent collision rates. Non-unique hashes, and the rate of change of content in caches and servers, can lead to false hits or "collisions". Having a collision-free hashing method, however, is not sufficient enough to prevent false hits. False hits can be the result of hashing collision in the routing table, but they can as well be the result of content aging. As previously mentioned in Section 2.6.1, a typical low cache miss rate is of the magnitude of 5% from the total number of requests [69]. Hence, it is important to note that the number of content aging misses is by magnitudes greater than the number of CRC32 collisions. A collision free signature method can guarantee no collisions in the routing table, however content aging misroutes can never be avoided completely, since digest updates will always delayed due to (at least) digest communication delay.

Storing multiple Bloom filters and implementing parallel "look-ups" into these filters is memory and CPU (hardware) intensive. The CRC32 digests are amenable to a direct hash-based look-up. Multiple routing tables can be merged into one look-up table. This is needed, because a single digest will contain a routing table for each distributed content source. For example, an implementation of a CRC32 signature based digest can be a hash table using a subset of the CRC32 codes directly. CRC32 is already well randomized and taking a subset of the 32-bits can be used instead of a hash function computation when each key is accessed. Hence, insertion of entries can be of the same complexity as a switching look-up.

Chapter Seven

Evaluation of Hashing Algorithms for URL Routing

In this chapter the Simple, H1, and Aggressive hashing algorithms are evaluated for their application to URL routing. This evaluation measures the hashing table look-up time and the behavior of a single server queue where the service center is the hashing table look-up. The application of the three hashing algorithms to URL routing is new and thus this evaluation chapter presents new insights on the relative performance of these algorithms. The evaluation of the queueing behavior of a hashing algorithm is an entirely new way of characterizing these types of algorithms. The criteria of interest in this evaluation are look-up and queueing delay.

7.1 Access lists used in this evaluation

For the evaluation of the simple, H1, and Aggressive hashing algorithms, two representative access logs that contained time-stamps (of access interarrival times) were used. The NASA [30] and Clark [29] access logs, were chosen due to their large size and use in other studies (that is, [4]). Tables 7.1 and 7.2 summarize the two access logs. In Table 7.1 the list size (1) is for the full URL string and list size (2) is for CRC32 signatures. The size of the CRC32 signature table is the number of URLs multiplied by four bytes. The NASA access list represented one month of activity and the Clark list one week of activity. These access logs are representative of requests in a CDN where content can be distributed to both servers and caches. The large size of the logs (many millions of accesses) and time of collection are significant.

Table 7.1 – Characterization of HTTP access lists for size

Access list name	# of accesses	# of URLS	Mean URL length	List size (1)	List size (2)
Clark	1,673,750	37,266	35.06 bytes	1,306,546 bytes	149,064 bytes
NASA	1,569,898	15,700	35.58	558,606	62,800

Table 7.2 – Characterization of HTTP access lists for time

Access list name	Duration	Mean interarrival time	Stddev interarrival time
Clark	605,776 sec	0.36 sec	5.62 sec
NASA	2,592,007	1.65	39.79

7.2 Description of evaluation method – look-up time

In Simple chain hashing an input key is hashed into a hash index. The hash index points to a chain of key-value pairs in the hash table. The input key is then compared to the stored keys in the hash chain and when a matching stored key is found, the value stored in association with that key is returned. For a hashing algorithm, the response variable of interest is the look-up time to find a value for a given input key. This look-up time is the product of the number of memory accesses and memory access time (the "speed" of the memory). In this evaluation, memory access time was normalized to 1.0. In a hash list a single look-up requires one or more memory accesses to find the value associated with a key. For example, if the value associated with a 32-bit key is the second entry in a Simple hash chain then two memory accesses are required: one access (and comparison) to determine that the head of the chain is not the key; and a second access (and comparison) to determine that the second entry is a match to the key. The value

associated with the second entry in the table is then read. The size of the key, its hash index, and associated value also affect the number of memory accesses. In this evaluation, a 32-bit wide memory was assumed. Thus, if a key was of size 32-bits only one memory access was needed to compare it to stored keys, and two memory accesses are needed to traverse and compare a single chain node. A larger key would require multiple memory accesses for comparison with the stored keys. A URL list that uses 32-bit signatures to represent the URL strings (the keys) reduces memory accesses when comparing keys for a match. Hash table size is measured in number of chains (in 2^K).

The three algorithms were implemented in the C programming language with a single main program and three called functions – each function a different hashing algorithm. Appendix A shows the data structure for a chain entry (key and value) and the look-up portion of the three hashing algorithms. The hash table was always stored in main memory using dynamically allocated memory (that is, from a C code `malloc()` statement). The main program took as input:

- Hash table size (*K*)
- File name of a URL list to populate the hash table
- File name of an URL access list of keys to look-up in the hash table

The URL lists existed in both URL string and signature formats. The URL access lists contained either a URL string or URL signature for the key. The program used the URL access lists to look-up values in the created hash tables where the number of memory accesses was counted for each key looked-up. The simulation ensures that all look ups were successful. That is, all looked up keys are known in advance and they are keys already stored in the hash. For a given hash table size, URL list, and URL access list the

minimum, maximum, mean, and standard deviation of the number of memory accesses were determined and reported.

7.3 Description of evaluation method – queueing behavior

A URL router can be modeled as a single server queue where the look-up is the service. Where hashing is used as the look-up method, the hashing algorithm defines the service center for the queue. Figure 7.1 shows a single-server queue where the service center is the hash table look-up (based on the number of accesses in the hash table) and the arriving customers are keys. The departing customers are the values resulting from the hash table look-up. The input to the queueing model are interarrival times (from the access list or exponentially distributed as described below) and number of memory accesses for a given URL associated with the interarrival time. The number of memory accesses was generated from the experiments in the previous section. This models a URL router where the hash table look-up is the service time for a request to be re-directed. The goal of the queueing study is to evaluate the performance of the hashing algorithms with respect to application in a queueing system – such as a URL router. This is a new way of studying the behavior of a hashing algorithm. The source code for this evaluation is using the CSIM18 simulation model. CSIM18 is a process-oriented simulation engine that is a C function library [67]. Thus, CSIM18 models are written in C. This source code models a trace-driven single-server queue. The `generate()` process reads the tuples from the trace file, holds for an interarrival time, and then queues the service demand (service time is proportional to the number of accesses) to the `queue1()` process. The `queue1()` process reserves, holds, and releases a server

facility. CSIM18 maintains statistics counters internally and the final statistics results are output at the end of the main program (named `sim()` in CSIM18 programs).

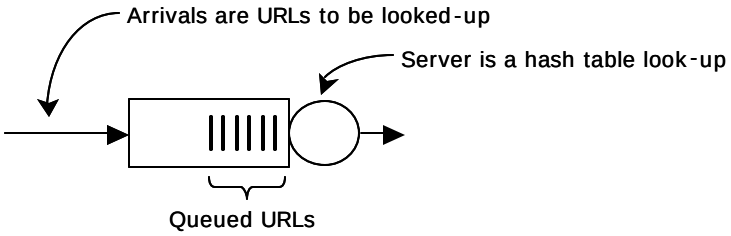

Figure 7.1 – Single-server queueing model for hashing algorithm evaluation

For a single server queue, the response variable of interest is queue length. Given a queue length and a known arrival rate, the queueing delay can be directly computed using Little's Law

$$W = \frac{L}{l} \tag{7.1}$$

where L is mean queue length, l is mean arrival rate, and W is mean wait or delay). The utilization of a queue, r (also U), is the arrival rate divided by service rate, m,

$$r = U = \frac{l}{m} \tag{7.2}$$

Queue length is affected by the mean, variance, and correlation properties of the customer interarrival times and service times. Buffer size was infinite in all simulation models. The control variables for the hashing algorithms evaluated for queueing behavior are the same as for the look-up time evaluation with the addition of the following:

- Utilization (U) of the server
- Request interarrival time distributed as such:
  - Exponentially distributed
  - Truncated maximum interarrival time ($T_{\max}$) from the URL access list
- Ordering of requests (unshuffled or shuffled)

The utilization of a server is controlled by the modeled time for a memory access. For a desired target utilization, U, the modeled memory access time, T_m, is determined as follows. For a given access list the total time for all accesses to arrive, T_{all}, is known (T_{all} is the sum of the interarrival times in the URL access list). The total number of look-ups, N_{total}, required to look-up all the keys in the access list is known from the previous look-up time evaluation. Then,

$$T_m = \frac{U \cdot T_{all}}{N_{total}} \tag{7.3}$$

where T_m is set as a control variable in the simulation model to achieve a desired server utilization, U. A method of truncating interarrival times was used to control the degree of burstiness of the interarrival times (measured by the coefficient of variation) in the URL access lists. By controlling the degree of burstiness, the queueing evaluation could be run for greater utilization levels. This method entailed setting a maximum interarrival time, $T_{\max}$. All interarrival times greater than $T_{\max}$ were truncated to $T_{\max}$. This resulted in reducing the burstiness of the interarrival times as measured by the coefficient of variation. Shuffling is used as a means of breaking long range dependence (LRD) in the order of accesses. Shuffling is implemented by selecting each entry in an URL access list

and swapping it with a randomly chosen entry in the list. This shuffling is repeated for several passes through the URL access list. LRD is well-known to affect queueing delay [39]. Autocorrelation is a measure of LRD. A slowly declining autocorrelation signifies the existence of LRD.

7.4 Evaluation of look-up time

The three hashing algorithms were compared for look-up time. The worst case in terms of hash look-up time will occur when the keys are accessed so that only the keys at the tail of the chain are accessed all the time. Most studies will not address this case, since its theoretical evaluation is trivial. The mean look up time will be examined next (again, in terms of the number of nodes traversed in a chain) to do a look-up. The NASA and Clark URL access lists (and associated URL lists), as summarized in Section 7.1 were used as input. The HTTP access and URL list was used in two formats – keys as URL strings and keys as 32-bit CRC32 URL signatures. The key control variable for the four algorithms was hash table size. The hash table size was varied as K=8, 9, ... 13. For each hash table created, the key properties of the hash table itself were measured. Tables 7.3 and 7.4 show the memory size and chain lengths statistics for K=8 (this value of K represents a dense table) for the NASA and Clark access lists.

Experiment #1: Evaluation of the effect of K on look-up time. Comparison of the three algorithms for K=8 to 13 for the NASA access list.

Experiment #2: Evaluation of the effect of K on look-up time. Comparison of the three algorithms for K=8 to 13 for the Clark access list.

Table 7.3 – CRC32 URL signatures populated hash table for NASA (K=8)

Algorithm	Memory size	Chain mean	Chain std dev	Chain min	Chain max
Simple	5.26 Mbytes	43.92	24.60	1.00	132.00
H1	4.53	3.25	12.32	1.00	132.00
Aggressive	2.69	1.74	7.46	1.00	132.00

Table 7.4 – CRC32 URL signatures populated hash table for Clark (K=8)

Algorithm	Memory size	Chain mean	Chain std dev	Chain min	Chain max
Simple	16.70 Mbytes	58.69	39.75	1.00	239.00
H1	0.39	15.94	31.30	1.00	239.00
Aggressive	0.21	9.45	20.26	1.00	239.00

7.4.1 Experiment results

The outcome from the experiments #1 and #2 are shown in Figures 7.2 and 7.3. For smaller hash table sizes (K=8 and K=10) resulting in densely populated hash tables Aggressive hashing outperforms Simple chain hashing by a factor of 16 and H1 by a factor of 1.7. For sparse hash tables (K>10) H1 and Aggressive hashing exhibit comparable look-up times. When the hash table size approaches the cardinality of the CRC32 space, 2^{32}, and the mean chain length is one, the performance of all hashing methods is expected to converge. Since the average worst case look-up time for the entire table equals the mean chain length (as explained in Section 5.3), when the mean chain length is one, the average number of look-ups will be one and the same for all

methods. Both Figure 7.2 and Figure 7.3 (each plotting a different input trace – Clark and NASA) show this trend for increasing values of *K*.

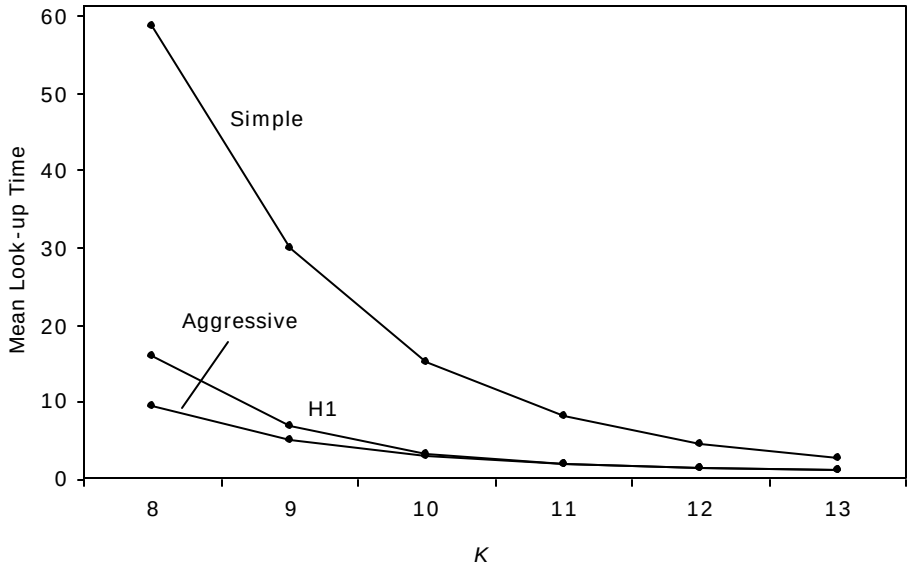

Figure 7.2 – Hash table look-up time for experiment #1

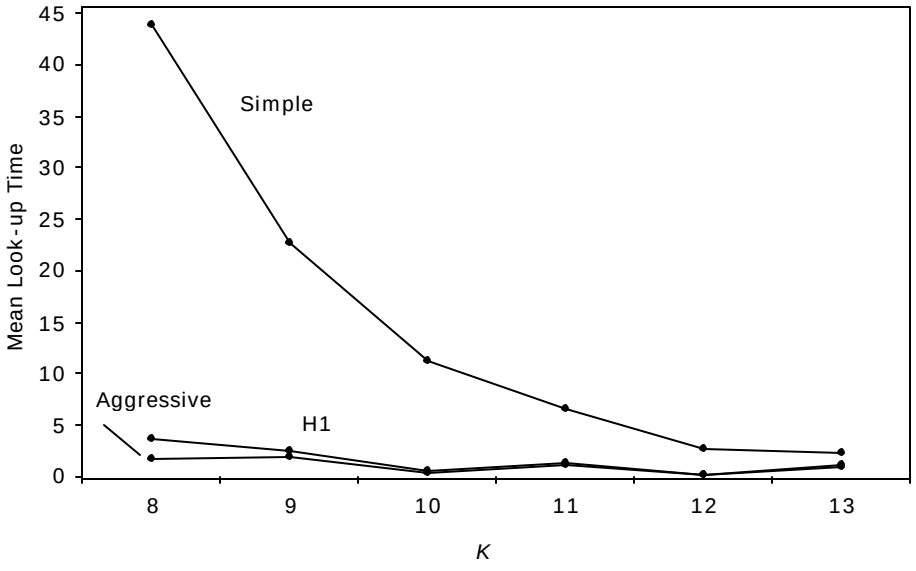

Figure 7.3 – Hash table look-up time for experiment #2

7.5 Evaluation of queueing behavior

The three hashing algorithms were compared for queueing delay. The HTTP access and URL lists were used only in signature format. For the queueing evaluation, only the NASA access list was used since the results in the previous section for the two lists were very similar. For the experiments, the control variables were hash table size (*K*), utilization of the queue (*U*), distribution of requests interarrival times, and ordering of requests. For distribution of interarrival times a maximum interarrival time ($T_{\max}$) is used in some experiments. In other experiments, the interarrival times are synthetically generated from an exponential distribution. The synthetically generated interarrival times replace the interarrival times taken from the access list in the trace file. The order of requests is used in two ways:

- As taken from the access list (unshuffled)
- Shuffled from the access list to break any long range dependence in request order

Figure 7.4 shows the autocorrelation for the NASA access list for the number of accesses for the three algorithms. The autocorrelation was computed as *r(k)* for a lag *k*.

$$r(k)\frac{E[(x_i - m)(x_{i-k} - m)]}{s^2}, \tag{7.4}$$

where x_i is the value that the random variable x takes at time i, m is the mean number of look-ups for the evaluated time series, and σ^2 is the variance of the number of look-ups. The results in Figure 7.4 show that Simple chain hashing and H1 hashing have a much higher autocorrelation than does Aggressive hashing.

Experiment #1: Evaluation of the effect of K on queue length for a fixed service rate such that Simple chain hashing is U = 80% and exponentially distributed interarrival times of requests. Mean queue length is measured.

Experiment #2: Evaluation of the effect of $T_{\max}$ on queue length for a fixed service rate such that Simple chain hashing is U = 80% and K = 8. Mean queue length and the utilization of H1 and Aggressive are measured.

Experiment #3: Evaluation of the effect of $T_{\max}$ on queue length for a fixed U = 80% and K = 8. Mean queue length is measured.

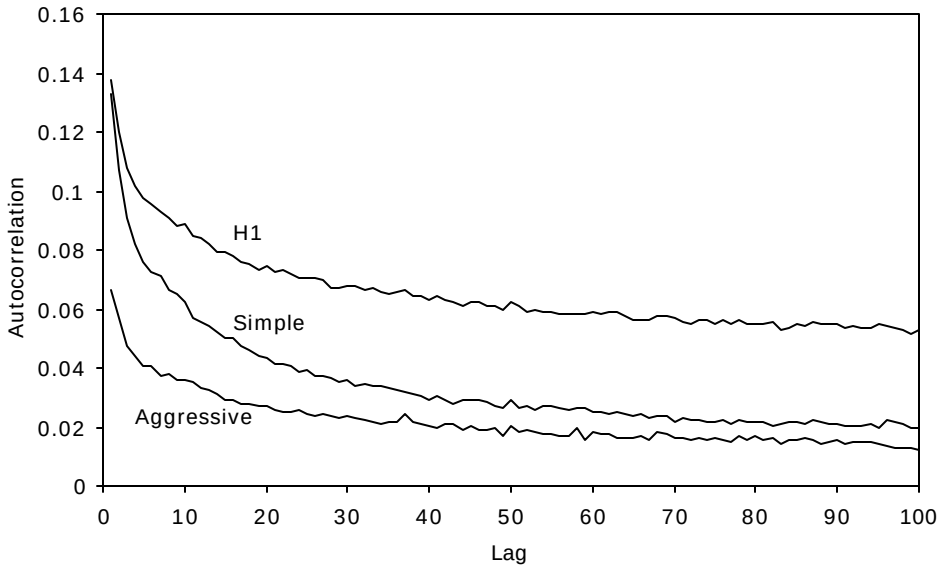

Figure 7.4 – Autocorrelation for 100 lags for number of accesses

Experiment #4: Evaluation of the effect of autocorrelation (unshuffled and shuffled ordering of requests) on queue length L for a fixed U = 80% and fixed hash table size K = 8. Mean queue length is measured.

Experiment #5: Evaluation of the effect of autocorrelation (unshuffled and shuffled ordering of requests) on queue length for a fixed service rate such that simple is U=80% and fixed hash table size K=8. Mean queue length is measured.

7.5.1 Experiment results

Results from Experiment #1 are shown in Figure 7.5. For smaller hash table sizes (K=8 and K=10), resulting in dense hash tables, Aggressive hashing outperforms the other methods: Simple chain hashing by a factor of 27 and H1 by a factor of about 2. H1

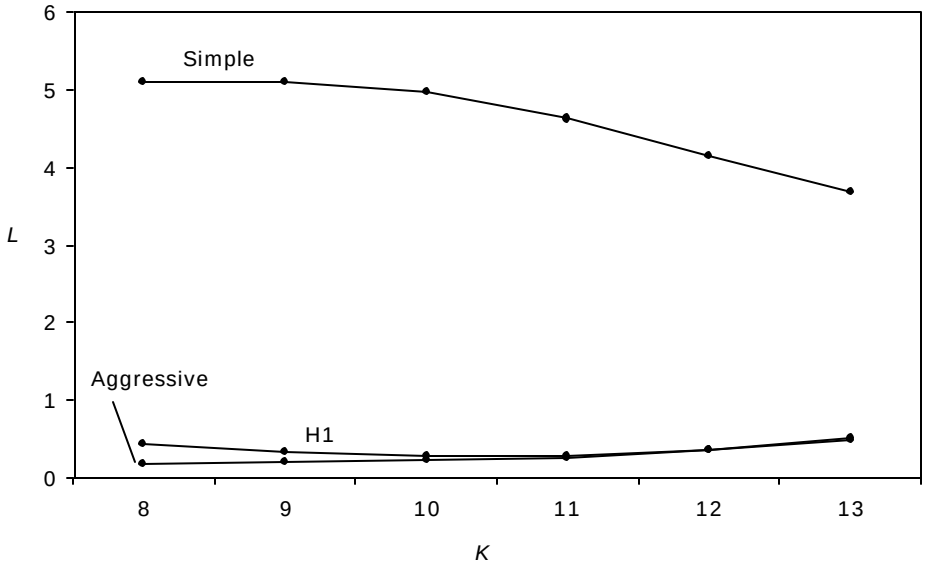

Figure 7.5 – The effect of hash table size (2^K) on mean queue length (L)

performs 13 times better than Simple chain hashing. When the mean chain length for all hash index positions gets closer to 2 the performance of all methods is close. With increase in K all methods converge as expected.

Results for experiment #2 are shown in Figure 7.6. Increase in burstiness levels for all methods resulted in increase in mean queue length. The mean queue length for Simple was magnitudes greater that that of H1 and Aggressive, ranging from 5500 to 34,000 requests. The observed utilization levels for H1 were about 22%, while Aggressive resulted in a drop down to 18%.

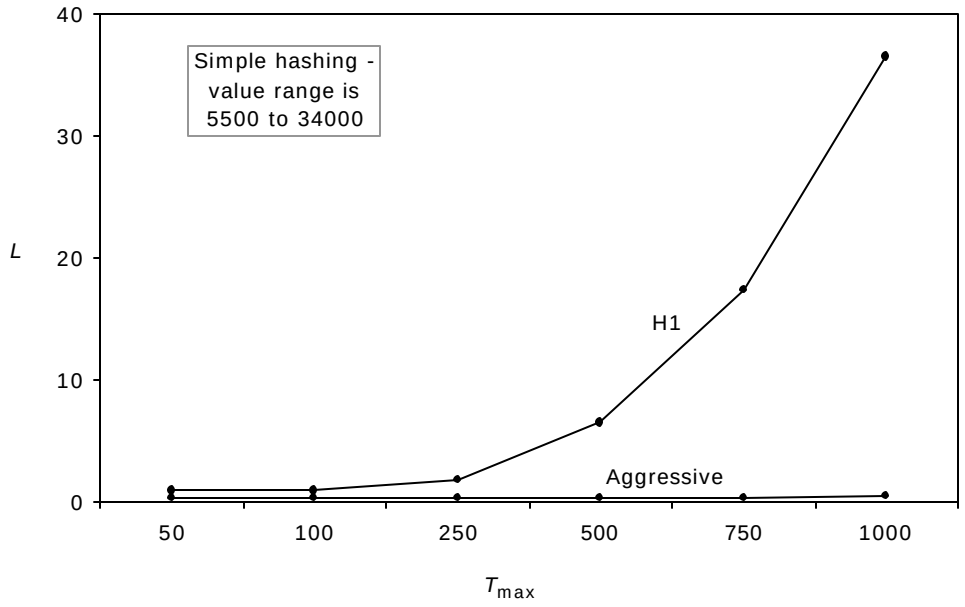

Figure 7.6 – The effect of burstiness ($T_{\max}$) on L for K=8

The results from experiment #3 are shown in Figure 7.7. H1 resulted in larger (by a magnitude) and faster increasing mean queue length than Simple and Aggressive hashing. This outcome is surprising, because H1 and Aggressive produce comparably smaller mean look-up delay as compared with Simple chain hashing, but the queueing

performance of Aggressive is much better than that of H1. This outcome indicates that there are other factors than mean number of look-ups that determine mean queueing delay and motivates the next two experiments.

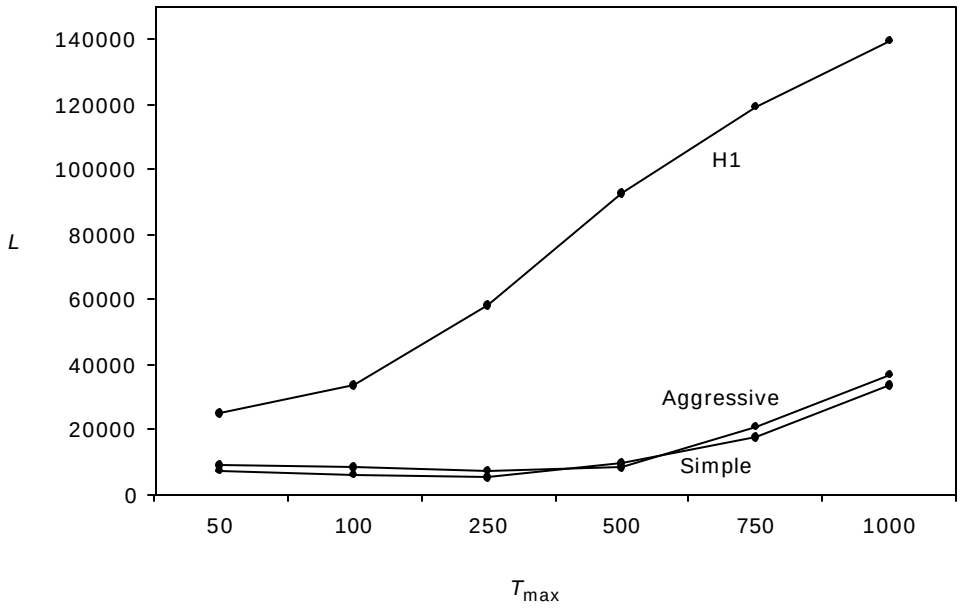

Figure 7.7 – The effect of $T_{\max}$ on L for K=8 and U=80%

Table 7.5 shows the results for experiment #4. The results include a theoretical M/G/1 queueing result for L using the Pollaczek-Khinchin formula [36]:

$$L = \frac{2r - r^2 + r^2 C_s^2}{2(1-r)} \tag{7.5}$$

where $r = U = l/m$ and C_s is the coefficient of variation of the service time. As seen in Figure 7.6 the M/G/1 results agree very closely with the shuffled results showing that the shuffling of the ordering resulted in an independent series.

Table 7.6 shows the results for experiment #5. The greater efficiency of H1 and Aggressive results in a much lower utilization than Simple chain hashing. H1 is about four times more efficient than Simple chain hashing (that is, four times less look-ups are needed to find a value) and Aggressive is a factor of 1.7 as efficient as H1. At the lower utilizations, mean queue lengths are very small as would be expected.

Table 7.5 – Mean queue length results for experiment #4

Algorithm	unshuffled	shuffled	M/G/1
Simple	5.20	3.15	3.13
H1	29102.01	8.58	8.57
Aggressive	294.09	9.93	9.76

Table 7.6 – Mean queue length results for experiment #5

Algorithm	*U*	unshuffled	Shuffled
Simple	80.0%	5.20	3.15
H1	21.7	0.43	0.36
Aggressive	12.9	0.19	0.18

7.6 Discussion of results

Aggressive hashing improves upon H1 hashing. The improvement in look-up time was modest, a factor of about two for both the evaluated traces. As expected with an increase in the hash table size, the mean hash chain length for all methods decreases. This result is intuitive, since a larger index space will result in decrease in mean number of collisions. The queuing performance of H1 and Aggressive hashing for sparse hash tables is also

similar. When the number of hash index locations is sufficiently large to prevent collisions of more than two keys at an index, both self-adjusting methods will have the same mean chain length and exactly the same look-up performance. A trend of merging the performance of these two methods with increase in table size was an expected observation since chain node rearrangement for chains shorter than two nodes in length is the same. Simple chain hashing will have the same performance as the other two methods if there are no collisions in the hash table. This will occur when the size of the index space matches the cardinality of the CRC32 space. Hence, a trend of merging the performance of all methods with increase in table size was an expected result, as well, since chain node rearrangement for chains shorter than two nodes in length is the same.

In the second set of queuieng evaluation experiments, where table size is fixed to match a densely populated table, and traffic burstiness varies, it was shown that using queueing as a means of evaluating hashing algorithms can be very insightful. This simulation–determined difference in mean look-up time between H1 and Aggressive hashing was 1.7 times for a fixed utilization of 80%. The difference in mean queueing delay was 6 times. When the requests were shuffled, all the hashing algorithms resulted in M/G/1 predicted queueing delay. This demonstrates that autocorrelation plays the role in queueing delay in excess of that predicted by the M/G/1 model and, very significantly, makes H1 hashing very unsuitable for use in routing tables. It is a remarkable result that only a very slight difference in mean look-up time can "hide" such a considerable difference in mean queueing delay. For any hashing implementation that is part of a queueing system, it is very important to evaluate queueing behavior. This dissertation has shown this for the first time.

Chapter Eight

Summary and Directions for Future Research

Content Distribution Networks (CDNs) are a rapidly growing service on the Internet. CDNs distribute and co-locate content to be geographically close to the users (of the content). This content mirroring 1) reduces the load on the origin server, 2) reduces traffic on the Internet, and 3) improves response time to the users. For CDNs to be feasible, methods of routing of HTTP requests originating from users are needed. URL routers need routing tables similar to IP routing tables in IP routers. However, the methods used for IP routing cannot scale to URL routing. In URL routing the number of destination addresses (URLs) greatly exceeds the number of IP addresses. In addition, URLs are not stable – new ones are constantly created and old ones disappear. The large size of URLs, their very large number, and their dynamic nature makes URL routing a difficult problem. This is the problem that is addressed in this dissertation.

In this dissertation, the problem of reducing the size and improving look-up performance of URL routing tables in URL routers is addressed. A URL router that uses HTTP redirection is architected in Chapter Four. The URL router front-ends a content server including the origin server site. As the front end, the URL router receives client HTTP requests by establishing a TCP connection with a requesting client. The HTTP request is either satisfied at the local site or re-directed with an HTTP redirection message to a better-to-the-user content source. This re-direction entails a routing look-up. Routing tables or other knowledge of location of content must be shared. The

location knowledge is the URL and its associated content source IP address(es). In Chapter Five methods of reducing the size of URL routing tables are investigated. Good methods will both reduce the size of the routing table and enable fast routing look-ups.

Cyclic Redundancy Check (CRC) codes are investigated as a means of reducing a URL string to a 32-bit integer signature. CRC32 can be computed reasonably efficiently in software and very efficiently in hardware. Properties of CRCs – that are used in this dissertation – make it possible for a single CRC hardware circuit to compute CRCs over multiple fields in a packet. Experiments show that CRC32 URL signatures result in collision rates comparable or equal to existing Bloom filter methods. Bloom filters cannot be updated in the same manner as can CRC32-based URL routing tables and are thus not applicable to URL routing. MD-5 signatures are guaranteed (with very high probability) to be unique at an expense of high computational complexity. This uniqueness property is lost when a Bloom filter is used. Thus, using CRC32 (with less computational complexity and "free" availability in existing network hardware) is motivated.

The CRC32 URL signatures lead naturally to the use of hash tables for implementing a URL routing look-up. CRCs are uniformly distributed random values where any sub-sequence of bits from a CRC is also uniformly random and thus can be used as "pre-computed" hash function value. The look-up delay and queueing behavior of the simple, H1, and new Aggressive hashing algorithms as applied to URL routing are investigated. It is shown that the Aggressive hashing algorithm has the best performance.

The main trade-off in implementing a URL router based on the CRC32 algorithm is that a device needs to be added to front end content sources in the CDN resulting in an

increase in initial investment cost to implement a CDN. CDNs also require maintenance. The analysis of cost-effectiveness is beyond the scope of this dissertation, but existing CDN services such as Akamai argue for their cost effectiveness.

8.1 Specific contributions from this research

This research has addressed an important and emerging problem in the Internet – how to route at an application or user layer. The four key contributions from this research as presented in this dissertation are:

1) A comprehensive examination of existing work in application layer routing and related areas (namely distributed caching structures). From this examination, a URL router was proposed and architected.
2) A new method of using CRC32 to reduce the size of URL tables was investigated and evaluated.
3) A new hashing method called Aggressive hashing was developed and evaluated. For dense hash tables this method has almost 2 times faster mean look-up times than the existing H1 hashing algorithm.
4) A new way to study the performance of hashing algorithms by evaluating their queueing behavior was pioneered. Surprising results were found in how the existing H1 hashing algorithm has very high autocorrelation in look-up times resulting in mean queue lengths two to three magnitudes greater than that of Aggressive hashing.

In summary, the evaluations performed in this dissertation show that CRC32 URL signatures and Aggressive hashing are significantly better methods for designing the next generation of URL routers to enable future growth of the Internet.

8.2 Directions for future research

This research has addressed how to build compact and efficient (in look-up) URL routing tables. What has not been addressed is how to associate a client request with the best content source. A look-up in a URL routing table can result in the IP addresses of multiple content sources that can satisfy the request. The geographic location of the client must be considered – how this is done is not addressed in this research – to determine the best content source to redirect the client to. Another issue not addressed in this research is the relationship between connection establishment and routing table look-up time in a URL router. One way to address this performance bottleneck may be to parallelize connection establishment and look-up. That is, have separate connection and look-up engines within a URL router. These are two areas for future research.

This research has shown that signatures can be used in place of URL strings to reduce the size of URL lists and enable fast hashing-based look-up of URL strings. This use of signatures to substitute for strings can have many applications beyond URL routing. For example, Napster-like peer-to-peer applications [64] that use a centralized directory for locating distributed content are in need of methods to reduce directory size and look-up time. Large directory size resulting in long look-up times caused a performance bottleneck at the centralized Napster server in 2000 [38]. Peer-to-peer networks without centralized directories that share directories are limited in scalability by the amount of

traffic they generate. Here again signature methods need to be investigated. A future direction of research is thus investigating the application of signatures to open problems in improving the scalability of peer-to-peer networks.

References

[1] Akamai Technologies, Inc. 2003. URL: http://www.akamai.com.

[2] The Apache Software Foundation, 2000. URL: http://www.apache.org.

[3] G. Apostolopoulos, V. Peris, P. Prashant, and D. Saha, "L5: A Self-Learning Layer-5 Switch," IBM Research Report RC 21461, April 1999.

[4] M. Arlitt and C. Williamson, "Web Server Workload Characterization: The Search for Invariants", *Proceedings of the ACM SIGMETRICS*, pp. 126-137, April 1996.

[5] Artificial Intelligence Research Group, Department of Computer Science and Engineering, University of Washington, URL: http://www.cs.washington.edu/ai/adaptive-data/.

[6] L. Aversa and A. Bestavros, "Load Balancing a Cluster of Web Servers Using Distributed Packet Rewriting," *Proceedings of the 2000 IEEE International Performance, Computing, and Communications Conference*, pp. 24-29, February 2000.

[7] J. Bentley, C. McGeoch, "Amortizing Analysis of Self-organizing Sequential Search Heuristics," *Communications of the ACM,*, Vol. 28, No. 4, pp. 404-411, 1985.

[8] A. Bestavros, "Demand-Based Document Dissemination to Reduce Traffic and Balance Load in Distributed Information Systems," *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing*, pp. 338-345, October 1993.

[9] B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Communications of the ACM*, Vol. 13, No. 7, pp. 422-426, July 1970.

[10] Cable News Network LP, 2003. URL: http://www.cnn.com.

[11] CCITT: Data Communication Networks – Open System Interconnection (OSI) Model and Notation, Service Definition; Blue Book Recommendations, X.200-X.219, International Telecommunication Union, 1989.

[12] F. Chung, D. Hajela, and P. Seymour, "Self-Organizing Sequential Search and Hilbert's Inequalities", *Journal on Computer Systems and Science*, Vol. 36, No.2, pp. 148-157, 1988.

[13] "Cisco Content Routing Protocols", white paper, 2000. URL: http://www.cisco.com/warp/public/cc/pd/cxsr/cxrt/tech/ccrp_wp.pdf.

[14] M. Crovella and R. Carter, "Dynamic Server Selection in the Internet," *Proceedings of the Third IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystem*s, pp. 158-162, June 1995.

[15] M. Dahlin, "Interpreting Stale Load Information," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 11, No. 10, pp. 1033-1047, October 2001.

[16] Data for Experiments, Department of Computer Science and Engineering, University of South Florida, URL: http://www.csee.usf.edu/~zgenova/traces/.

[17] DOD standard "Internet Protocol," RFC 760, January 1980. URL: http://www.ietf.org/rfc/rfc760.txt.

[18] J. Elson and A. Cerpa, "Internet Content Adaptation Protocol (ICAP)", RFC 3507, April 2003.

[19] L. Fan, P. Cao, J. Almeide, and A. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *Proceedings of ACM SIGCOMM*, pp. 254-265, September 1998.

[20] Z. Fei, S. Bhattacharjee, E. Zegura, and M. Ammar, "A novel selection technique for improving the response time of a replicated service," *Proceedings of INFOCOM*, pp. 783 - 791, 1998.

[21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee "Hypertext Transfer Protocol -- HTTP/1.1," RFC 2068, January 1997.

[22] S. Gai, "Content Delivery Networks," presentation at *TERENA Network Conference*, 2000. URL: ftp://ftpeng.cisco.cin/sgai/t2000cdn.pdf.

[23] Z. Genova and K. Christensen, "Challenges in URL Switching for Implementing Globally Distributed Web Sites," *Proceedings of the Workshop on Scalable Web Services*, pp. 89-94, August 2000.

[24] Z. Genova and K. Christensen, “Efficient Summarization of URLs using CRC32 for Implementing URL Switching,” *Proceedings of the 27th IEEE Conference on Local Computer Networks (LCN)*, pp. 343-344, November 2002.

[25] Z. Genova and K. Christensen, "Managing Routing Tables for URL Routers in Content Distribution Networks," submitted to the *International Journal of Network Management* in June 2003.

[26] Z. Genova and K. Christensen, “Using Signatures to Improve URL Routing,” *Proceedings of IEEE International Performance, Computing, and Communications Conference*, pp. 45-52, April 2002.

[27] J. Gwertzman and M. Seltzer, “The Case for Geographical Push-Caching,” *Proceedings of the Fifth Annual Workshop on Hot Operating Systems*, pp. 51-55, May 1995.

[28] C. Hendricks, “Routing Information Protocol,” 1988. URL: http://www.ietf.org/rfc/rfc1058.txt.

[29] HTTP requests to the ClarkNet Web server, URL: ftp://ita.ee.lbl.gov/traces/.

[30] HTTP requests to the NASA Web server, URL: ftp://ita.ee.lbl.gov/traces/.

[31] Inktomi Corporation, 2003. URL: http://www.inktomi.com.

[32] D. Irvin, "Preserving the Integrity of Cyclic-Redundancy Checks when Fully Protected Text is Intentionally Altered," *IBM Journal of Research and Development*, Vol. 33, No. 6, pp. 618 - 626, November 1996.

[33] K. Johnson, J. Carr, M. Day, and F. Kaashoek, “The Measured Performance of Content Distribution Networks,” *Computer Communications*, Vol. 24, No. 2, pp. 202-206, February 2001.

[34] J. Kangsharju, K. Ross, and J .Roberts, “Performance Evaluation of Redirection Schemes in Content Distribution Networks”, *IEEE Journal on Computer Communications*, Vol. 24, No. 2, pp.207-214, 2001.

[35] D. Katabi and J. Wroclawsi, “A Framework for Scalable Global IP-Anycast (GIA)”, *Proceedings of ACM SIGCOMM*, pp. 3-15, September 2000.

[36] L. Kleinrock, *Queueing Systems: Theory, Volume 1*, John Wiley & Sons; 1975.

[37] D. Knuth, *The Art of Computer Programming: Sorting and Searching*, Vol. 3, Second Edition, Addison Wesley Longman, 1998.

[38] J. Kurose and K. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*, Second Edition, Addison Wesley, 2003.

[39] W. Leland, M. Taqqu, W. Willinger, and D. Wilson, "On the Self-Similar Nature of Ethernet traffic (Extended Version)," *IEEE/ACM Transactions on Networking*, Vol. 2, pp. 1-15, February 1994.

[40] MCSNet (Winstar), URL: http://www.mcs.net/~www/lib-http/logs/OLD.

[41] B. Michel, K. Nikoloudakis, P. Reiher, and L. Zhang, "URL Forwarding and Compression in Adaptive Web Caching," *Proceedings of IEEE INFOCOM*, pp. 670-678, March 2000.

[42] Microsoft Corporation, 2003. URL: http://www.microsoft.com.

[43] M. Mitzenmacher, "How Useful is Old Information? [Load Balancing in Dynamic Distributed Systems]," *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pp. 83-91, August 1997.

[44] P. Mockapetris, "Domain Names – Concepts and Facilities", RFC 882, November 1983.

[45] J. Moy, " OSPF Version 2", July 91. URL: http://www.ietf.org/rfc/rfc1247.txt.

[46] Monthly Log Files 2000, Computer Science Division, University of California, Berkeley. URL: http://www.cs.berkeley.edu/logs/http/.

[47] K. Moore, "SONAR – A network proximity service version 1," draft-moore-sonar-03.txt, Internet Draft, Network Working Group, August 1998.

[48] Netscape.com, 2003. URL: http://www.netscape.com.

[49] NLANR Sanitized Cache Access Logs (National Science Foundation grants NCR-9616602 and NCR-9521745), 2000. URL: ftp://ircache.nlanr.net/Traces/.

[50] V. Pai, M. Aron, B. Gurav, M. Svendsen, H. Druschel, W. Zwaenepoel, and E. Nahum, "Locality-Aware Request Distribution in Cluster based Network Servers," *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 205-216, San Jose, October 1998.

[51] L. Pagli, "Self-adjusting hash tables," *Information Processing Letters*, Vol.21, Iss. 1, pp. 23-25, July 1985.

[52] PEAK Internet Service Provider and Education Center. URL: http://www.peak.org/http/real/logs/.

[53] PKZIP utility, PKWARE, Inc., 2000.

[54] D. Plummer, "Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48 bit Ethernet Address for Transmission on Ethernet Hardware," RFC 826, November 1982.

[55] Press Release, The Associated Press, Oct 11 1999.

[56] R. Rivest, "On Self-Organizing Sequential Search Heuristics," *Communications of the ACM*, Vol. 19, pp. 63-67, 1976.

[57] R. Rivest, "The IP Network Address Translator (NAT)," RFC 1631, May 1994.

[58] R. Rivest, "The MD5 Message Digest Algorithm," RFC 1321, April 1992.

[59] A. Roginsky, K. Christensen, and S. Polge, "Efficient Computation of Packet CRC from Partial CRCs with Application to the Cells-in-Frames Protocol," *Computer Communications*, Vol. 21, No. 7, pp. 653-661, June 1998.

[60] A. Rousskov and D. Wessels, "Cache Digests," *Computer Networks and ISDN Systems*, Vol.30, No.22-23, pp. 2155-2168, November 1998.

[61] San Diego Music Awards, 2001, URL: http://ksdm.com/sdma.

[62] Sandpiper Network, 1998. URL: http://www.sandpiper.com.

[63] Sanitized Log Files from Canada's Coast to Coast Broadband Research Network (CA*netII), 2000, URL: http://ardnoc41.canet2.net/cache/squid/rawlogs/.

[64] S. Saroiu, P. K. Gummadi, and S. D. Gribble, "A measurement study of peer-to-peer file sharing systems," *Proceedings of Multimedia Computing and Networking*, San Jose, CA, January 2002.

[65] D. Sarwate, "Computation of Cyclic Redundancy Checks via Look-up Table," *Communications of the ACM*, Vol. 31, No. 8, pp. 1008-1013, August 1988.

[66] B. Schneier, A*pplied Cryptography*, Second Edition, John Wiley & Sons, 1996.

[67] H. Schwetman, "CSIM18-The Simulation Engine," *Proceedings of the 1996 Winter Simulation Conference*, pp. 517-521, December 1996.

[68] D. Sleator, R. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *Communications of the ACM,*, Vol. 28, No. 2, pp. 202-208, 1985.

[69] K. Sripanidkulchai, B. Maggs, and H. Zhang, "Enabling Efficient Content Location and Retrieval in Peer-to-Peer Systems by Exploiting Locality in Interests", *Computer Communication Review,* Vol. 32 , Iss. 1, pp. 80-80, January 2002.

[70] W. Stallings, *Handbook of Computer-Communications Standards; Vol. 1: The Open Systems Interconnection (OSI) Model and OSI-Related Standards*, Macmillan Publishing Co., Inc., Indianapolis, 1987.

[71] Strong Cache Consistency for the Web, Local and Distant Server Logs at Virginia Tech, URL: http://www.cs.wisc.edu/~cao/icache/proxytrace.html.

[72] W. Tang, F. Du, M. W. Mutka, L. Ni, and A. Esfahanian, "Supporting Global Replicated Services by a Routing-Metric-Aware DNS," *Proceedings of the 2nd International Workshop on Advanced Issues of E-Commerce and Web-Based Information System*, pp. 67-74, June 2000.

[73] J. Touch, "Performance Analysis of MD5," *Proceedings of ACM SIGCOMM*, pp. 77 - 86, September 1995.

[74] "Transmission Control Protocol," RFC 793, September 1981.

[75] V. Valloppillil and K. Ross, "Cache Array Routing Protocol (CARP)," Version v1.0, Internet Draft, Microsoft Corporation, February 1998, URL: http://icp.ircache.net/carp.txt.

[76] D. Wessels and K. Claffy, "Internet Cache Protocol (ICP) Version 2," RFC 2186, September 1997.

[77] D. Wessels, A. Rousskov, H. Nordstrom, and A. Chadd, "Squid Web Proxy Cache," URL: http://www.squid-cache.org/.

[78] Yahoo! Incorporated, 2003. URL: http://www.yahoo.com.

[79] L. Zhang, S. Michel, K. Nguyen, A. Rosenstein, S. Floyd, and Van Jacobson, "Adaptive Web Caching: Towards a New Global Caching Architecture," *Computer Networks ISDN Systems*, Vol. 30, pp. 2169-2177, November 1998.

Appendices

Appendix A

Source Code for Hashing Algorithms

This appendix contains the source code for the simple, H1, and Aggressive hashing algorithms as described in Chapters Two and Five of this dissertation. The data structures for each algorithm are presented first and then the main loop for look-up in the hash table is presented.

A.1 Data structure for Simple chain hashing

```
struct entry                  // Hash table entry structure
{
  unsigned int key;           // ** key
  unsigned int value;         // ** associated value for key
  struct entry *next_ptr;     // ** pointer to next entry in chain
};
```

A.2 Data structure for H1 and Aggressive hashing

```
struct entry                // Hash table entry structure
{
      unsigned int key;         // key
      unsigned int value;       // associated value for key
      struct entry *next_ptr;   // pointer to next entry in chain
      struct entry *prev_ptr;   // pointer to previous entry in chain
};
```

A.3 Main loop for Simple chain hashing look-up

```
// Get CRC32 values from stdin, clip CRC32s, and then do a look-up
hit_count = miss_count = 0;
while(1)
{
  // Read the CRC32
  scanf("%08X \n", &crc32);

  // Initialize chain length
  c=0;

  // Clip the CRC32 to get a hash index
  hash_index = mask & crc32;

  // Do the look-up (and increment hit or miss counter appropriately)
  temp = &table[hash_index].key;
```

Appendix A: (Continued)

```
  while(1)
  {
      if (temp->key == crc32)
      {
        hit_count++;
        break;
      }

      if (temp->next_ptr == NULL)
      {
        miss_count++;
        break;
      }
      temp = temp->next_ptr;
      c++;
  }

  // Bail-out if EOF
  if (feof(stdin)) break;
}
```

A.4 Main loop for H1 hashing look-up

```
// Get CRC32 values from stdin, clip CRC32s, then do a look-up
hit_count = miss_count = 0;
while(1)
{
  // Read the CRC32
  scanf("%08X \n", &crc32);

  // Initialize chain length
  c=0;

  // Clip the CRC32 to get a hash index
  hash_index = mask & crc32;

  // Do the look-up (and increment hit or miss counter)
  temp = &table[hash_index].key;
  while(1)
  {
    if (temp->key == crc32)
    {
      hit_count++;
      if (temp->prev_ptr != NULL)
      {
```

Appendix A: (Continued)

```
        // Swap entry for bubble up
        swap_key = temp->key;
        swap_val = temp->value;
        temp->key = temp->prev_ptr->key;
        temp->value = temp->prev_ptr->value;
        temp->prev_ptr->key = swap_key;
        temp->prev_ptr->value = swap_val;
      }
      break;
    }

    if (temp->next_ptr == NULL)
    {
      miss_count++;
      break;
    }
     temp = temp->next_ptr;
     c++;
  }

  // Bail-out if EOF
  if (feof(stdin)) break;
}
```

A.5 Main loop for Aggressive hashing look-up

```
// Get CRC32 values from stdin, clip CRC32s, and then do a look-up
hit_count = miss_count = 0;
while(1)
{
  // Read the CRC32
  scanf("%08X \n", &crc32);

  // Initialize chain length
  c=0;

  // Clip the CRC32 to get a hash index
  hash_index = mask & crc32;

  // Do the look-up (and increment hit or miss counter appropriately)
  head = temp = &table[hash_index].key;

  while(1)
  {
    if (temp->key == crc32)
    {
      hit_count++;
      if (temp->prev_ptr != NULL)
      {
```

Appendix A: (Continued)

```
         // Move entry to head
         if (temp->next_ptr != NULL)
         {
            temp->next_ptr->prev_ptr = temp->prev_ptr;
            temp->prev_ptr->next_ptr = temp->next_ptr;
         } // not end of chain
         else // end of chain
         {
            if (table[hash_index].next_ptr != temp)
               temp->prev_ptr->next_ptr=NULL;
            else
            {
               key=table[hash_index].key;
               val=table[hash_index].value;
               table[hash_index].key=temp->key;
               table[hash_index].value=temp->value;
               temp->key=key;
               temp->value=val;
               break;
            }
         } // end not head

         key=table[hash_index].key;
         val=table[hash_index].value;
         table[hash_index].key=temp->key;
         table[hash_index].value=temp->value;
         temp->key=key;
         temp->value=val;
         temp->prev_ptr = &table[hash_index];
         temp->next_ptr = table[hash_index].next_ptr;
         table[hash_index].next_ptr->prev_ptr=temp;
         table[hash_index].next_ptr=temp;
       }
        break;
    }  // end hit case
    if (temp->next_ptr == NULL)
    {
      miss_count++;
      break;
    }
    temp = temp->next_ptr;
    c++;
  }

  // Bail-out if EOF
  if (feof(stdin)) break;
}
```

About the Author

Zornitza Genova Prodanoff received her bachelor's degree in Management Information Systems from the University of South Florida. She holds a master of science in Computer Science from the same university, where she worked as a research assistant in the field of computer networks. Her Ph.D. work was funded by the National Science Foundation under a grant of her advisor Dr. Kenneth J. Christensen. She is the author, and coauthor of several book chapters and articles on topics including performance evaluation of computer networks, time-constrained multimedia presentations, database design, and network programming. Her concern for education has helped her to hold several teaching assistantship positions within the colleges of Engineering and Business Administration at the University of South Florida. Her professional affiliations include IEEE, ACM, ASEE, and IEEE Women in Engineering.