Using Signatures to Improve URL Routing

Z. Genova and K. J. Christensen
Department of Computer Science and Engineering
University of South Florida, Tampa, FL 33620
{zgenova, christen}@csee.usf.edu

Abstract

Devices which forward or route at layers higher than the IP layer are emerging to support growing World Wide Web Content Distribution Networks (CDNs). URL routing can be used to forward or redirect a client HTTP request to the nearest or otherwise best content source. We develop and evaluate a new look-up method that uses fixed-length URL signatures based on CRC32 coding of URLs. The use of URL signatures results in smaller update messages, smaller routing tables, and faster look-ups for URL routers. We show how the circuit used to generate an Ethernet packet CRC32 can be simultaneously used to generate the CRC32 of a URL within an HTTP request. We evaluate signature collisions, CPU resources needed to generate signature URL lists, URL list size, look-up performance, and TCP connection performance using nine representative server and cache traces. Signature collisions occur for a negligible number of URLs. Signature-based URL lists consume ten times less memory and have look-up rates five to ten times faster than when using full-URLs.

1. Introduction

Content is becoming distributed throughout the Internet in large-scale Content Distribution Networks (CDN's). A single origin server site contains the "original" content. This content is then co-located throughout the Internet in *content sources* such as distributed servers and caches. Distribution into content sources can occur by both "push" (e.g., to distributed servers) or by "pull" (e.g., into reverse, transparent, and proxy caches). It is necessary to be able to coordinate between these content sources to ensure that the state of content is known. Each object in a content source may be represented by a Uniform Resource Locator (URL). URLs are variable length strings and are typically several tens of characters in length. HTTP requests which contain URL strings are sent by clients to content sources, which may respond with the requested object.

Caching of Web content 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. An HTTP request sent to a cache may result in a miss, in which case the cache obtains the requested content from another cache or the origin server (and then responds to the original request). Distributed caching infrastructures, such as Summary Cache [7], use MD5 signatures of URL strings reduced into Bloom filters to share knowledge of cache contents (i.e., the currently stored objects in the form of a list of URLs). In this manner, a request sent to a cache device that does not contain the object can be requested by the first cache device from another cache device that is known to contain the object. The use of Bloom filters reduces the size of the control messages comprising the URL lists sent between caches at the expense of an occasional "false hit". A false hit occurs when, 1) two or more URLs reduce into the same bit pattern in the Bloom filter which results 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 to and an update message had not yet been sent by this cache. False hits and misses are handled in the same way.

Commercial CDNs use provider-owned servers distributed throughout the Internet. The Akamai service [12] pushes origin server contents into Akamai-owned servers based on localizing of user requests. A user request is made to the origin site which returns an "Akamized" HTML document with embedded links pointing to the nearest Akamai server(s). Typically, the GIF and JPG images constituting the embedded links are served by Akamai-owned servers thus reducing load on the origin server. A large overhead is required to maintain state information about the distributed content and to determine to which Akamai server a given client should be directed (via the embedded links).

Routing at the URL level is currently a topic of much interest for research [1, 2, and 9]. URL routers are

This material is based upon work funded by the National Science Foundation under Grant No. 9875177.

intended to reduce the load on content sources allowing them to serve content only and not to handle routing at the same time. URL routing is much more difficult than IP routing for several reasons. The cardinality of URLs at a single site is much higher than that of IP addresses (a single cache will have one IP address but potentially thousands to millions of URLs). URLs at many sites (e.g., at caches) are transient lasting only minutes or hours. URLs are of variable length and are much longer than four-byte IPv4 addresses. URL routing also requires handling of TCP connection semantics. In this paper, we address how to 1) reduce the size of URL lists, and 2) improve look-up methods within a URL router.

The remainder of this paper is organized as follows. Section II describes the operation and architecture of a URL router. Section III describes how CRC32 URL signatures can be used for improving the performance of URL routing. An experimental performance evaluation of URL signatures is presented in Section IV. Section V summarizes and describes future work. Two appendices describe prefix look-ups and URL CRC32 generation.

2. The URL Router

A URL router makes forwarding decisions based on the URL contained in an HTTP request. URL routers are often called layer-5 or layer-7 switches [2]. A URL router can front-end an origin site (containing one, or a cluster of, servers), front-end a caching infrastructure, or serve as a proxy device on the client side. Figure 1 shows the possible locations of a URL router relative to caches and servers.

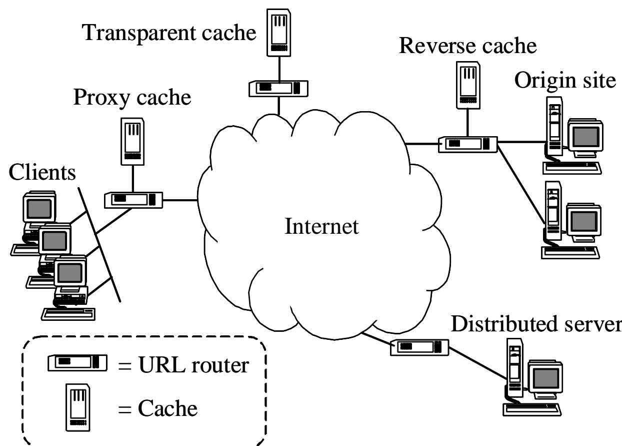

Figure 1. Placement of URL routers in the Internet

There are three different mechanisms that a URL router can use to route or forward HTTP requests:

1. DNS redirection [4]
2. TCP connection spoofing [20] or splicing [13]
3. HTTP redirection [8]

The HTTP 30X mechanism can be used directly by a server to send an HTTP redirection response message to a client. This redirection message instructs the client browser to automatically re-request the originally requested object at another host. 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 HTTP request must be processed through all layers of the protocol stack and be handled directly by the server software. Generating an HTTP redirection response can require the same workload (on a possibly overloaded server) as serving the requested content [19]. This further motivates the need for URL routers to front-end content sources.

2.1 Operation of a URL router

The semantics of an HTTP request require that a TCP connection be established before the request is sent. DNS redirection can be used to redirect the TCP SYN packet directly. For TCP connection spoofing/splicing or HTTP redirection, a connection must be established. A URL router must have an IP address that represents the content source or is the proxy address. Following connection establishment, a URL router can maintain the TCP connection to the client and spoof or splice the connection to a local (i.e., within the same LAN) server or cache. Or, the URL router can return an HTTP redirection message to the client and then terminate the connection. HTTP redirection automatically forces the client to resend its request to a new content source as specified in the HTTP redirection message.

The first generation of URL routers [1, 2] was used to front-end content sources and forward incoming client requests to the best server or cache in a local cluster. The best content source could be determined by dynamic metrics such as server load or static metrics such as matching the server type to the requested content type (e.g., forwarding requests for streaming video to a video server). TCP connection spoofing or splicing was used to maintain the client connection established to the URL router. The second generation of URL routers [9] 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 forced the client to re-request the content from another server site. Figure 2 shows the flow of a request

being (1) forwarded or (2) redirected (DNS redirection is not shown here). Methods for determining the content source to which a request should be redirected or forwarded are covered later in this paper. In this paper, we focus on HTTP redirection, but the methods developed apply equally to DNS redirection and TCP connection spoofing/splicing.

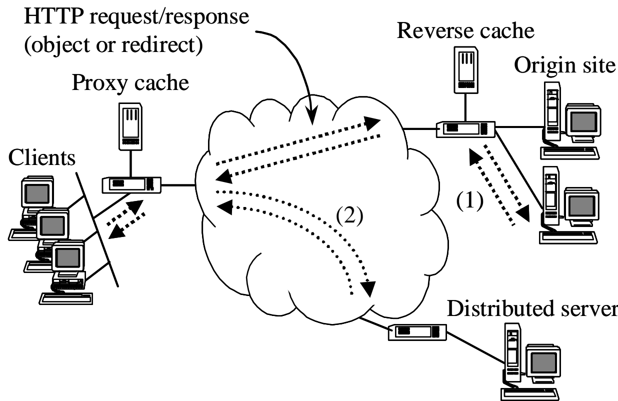

Figure 2. URL router (1) forwarding (2) redirecting

2.2 Architecture of a URL router

A URL router can be implemented with a general purpose PC or workstation attached to a layer 3/4 switch (e.g., as a "one armed" router), or with specialized hardware within a layer 3/4 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 is maintained in a global memory and transferred fully or partially to local memory on the line cards. Switching decision are made by the line cards whenever possible or, if not possible, by a central CPU. For a URL router implemented within a PC as a one armed router, the layer 3/4 switch forwards to the URL router TCP connection requests (i.e., SYN packets) designated for the IP address of the represented content source. Figure. 3 shows the one armed URL router.

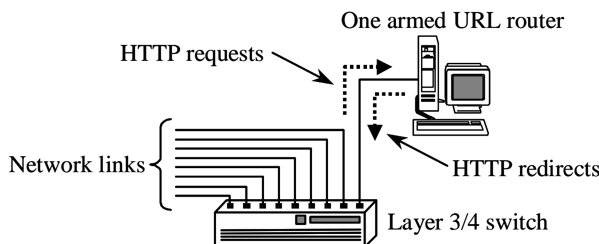

Figure 3. One-armed URL router

A URL router, whether implemented in a general purpose PC or as a specialized device, 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 hostname 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 also maintains a URL routing table from updates received from other URL routers or directly from content sources.

2.3 Structure of a URL routing table

A URL routing table is used to determine where to forward or redirect a request. The table contains URLs and the locations of content sources. Associated with every content source is the load and distance state. This state information, along with knowledge of the client's location, is used to select the best content source using load balancing methods such those as described in [5] and [10]. Fig. 4 shows the structure of a routing table with N URLs and M_i ($i = 1, 2, \ldots, N$) locations per URL. Except at the origin server, 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 transmit the URL lists and to reduce the processing complexity needed to perform look-ups.

URL 1	Loc 1 (state), loc 2 (state), … loc M_1 (state)
URL 2	Loc 1 (state), loc 2 (state), … loc M_2 (state)
URL 3	Loc 1 (state), loc 2 (state), … loc M_3 (state)
⋮	⋮
URL N	Loc 1 (state), loc 2 (state), … loc M_N (state)

Figure 4. Structure of a URL routing table

Content stored in caches is likely independent of origin server directory structures. However, content in distributed servers may be pushed and thus may contain entire directory structures of thousands of files. Appendix A describes prefixing and the use of multiple look-ups.

2.4 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-up (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 out of disk storage, look-up times will be greatly increased.

For a specialized URL router device, a single-line card will likely contain two CPUs (or one CPU and one special-purpose ASIC). 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 next section, we address the memory constraints and CPU bottlenecks for URL look-up. We focus primarily on request redirection and less on connection spoofing or splicing.

3. Using URL Signatures

We propose to use 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 [2] the entire URL is stored as the key value. 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.

3.1 Generating and using URL signatures

For a signature algorithm we have chosen to use Cyclic Redundancy Check (CRC) codes. 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 and is an integral part of all network adapters. At the URL router, all incoming HTTP requests have their URLs 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. Appendix B describes how this can be done.

The CRCs that are generated on-the-fly when an HTTP request is received are then used in the URL routing table look-up. For a simple chained hash table, the 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.

3.2 Prevention and handling of false hits

A false hit will occur when CRC signatures of different URLs are not unique. This CRC "collision" may result in an HTTP request being redirected to a content source that does not contain the requested object. In the case of redirection to a cache, the cache will itself request the object (e.g., from the origin site) and store it for future redirections. In the case of redirection to a distributed server, there are two ways to handle mis-routing due to CRC collisions. One method is to prevent collisions by checking CRCs during file generating and forcing a file renaming if a collision occurs. A second method is to use HTTP redirection to force another redirection to the origin site. This double re-direction will occur only very rarely.

The probability of two URLs hashing to the same CRC is, in the best case, the same as the birthday paradox. The best case occurs when the CRC function generates uniformly random hashes. For 2^K unique hashes (K = 32 for CRC32) and N URLs, ,

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

For a given URL in a list of N URLs,

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

And, the expected (mean) number of collisions in a set of N URLs is then approximately,

$$E[\text{number of collisions}] = N\left(1 - \left(\frac{2^K - 1}{2^K}\right)^{N-1}\right) \quad (3)$$

Table 1. Summary statistics for the access lists

Access list name	Number of accesses	Number of URLs	Mean URL length	URL list size (full URL)	URL list size (CRC32)	Collisions in URL list	Collisions in access list
www.peak.org [18]	16,374	70	23.93 bytes	1,675 bytes	280 bytes	0	0
SDMA [22]	41,941	153	33.76	5,165	612	0	0
UVA [25]	318,899	45,816	44.91	2,057,625	183,264	0	0
NLANR [17]	944,028	504,967	58.44	29,510,135	2,019,868	68	114
UC Berkeley [16]	1,791,349	149,344	41.87	6,253,716	597,376	2	14
mcs.net [14]	1,862,070	75,361	29.87	2,250,829	301,444	0	0
hyperreal.org [3]	4,080,590	86,338	89.17	7,698,337	345,352	2	2
CA*netII [23]	4,642,861	2,552,045	57.83	147,573,556	10,208,184	1,558	2,749
USF CSEE [6]	8,819,454	49,029	51.84	2,541,483	196,116	2	123

4. Evaluation of URL Signatures

We evaluate the use of URL signatures for URL routing in terms of four criteria:

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

We also briefly compare the look-up performance of URL table look-ups to TCP connection overhead.

4.1 Access lists used in the evaluation

To evaluate the performance improvements possible for using URL signatures, we used representative access lists based on seven server and two proxy cache logs. 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 is stored as a simple chained hash table with either:

1. Full URL as the key
2. Compressed (CRC32) URL as the key.

For each key, a four-byte value is stored emulating an IP address of a content source in a URL routing table. The URL list hash table is of length 2H entries (H = number of bits in the hash value taken as the first H bits of the URL CRC32 and). For the hash table with full URLs as keys, the hash values are still taken as the first H bits of the URL CRC32. The value of H is a control variable in the performance measurement. A larger H results in greater memory usage, but smaller hash chain with faster look-ups.

Table I summarizes the access logs. Table I shows the mean URL length in each access list and the size of the URL list with full URLs and with CRC32 signatures. The compressed 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). The Table I also shows the number of CRC collisions for the access and URL lists. These results are further described in the next section.

4.2 Evaluation of false hits (CRC collisions)

The first experiment evaluated the number of false hits (or CRC collisions) in the access and URL lists. A CRC32 was generated for each URL and the number of non-unique CRC32s counted. Table I shows the collision results 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 collisions for access lists. Table II shows URL list collision probabilities and the expected number of collisions (from Eq. 3). CRC32 signatures appear to be uniformly random with the measured and expected number of collisions close to the same.

Table 2. CRC32 URL list collision probability

Access list name	Collisions measured	Expected value (Eq.3)	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

4.3 Evaluation of the cost of signature generation

For the second experiment, we evaluated the CPU resource cost of generating 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 (similar to [24]). The total time required to generate a CRC32 in memory for each URL in the URL list was measured on an 866-Mhz Dell Dimension PentiumIII with Windows2000. In Table III CPU measurements are to a precision of 10 milliseconds. Ongoing work [11] shows that CRC32 compressed URL lists require six times less CPU resources than MD5-Bloom filter methods [7] and have the same or better collision rates. We show that CRC32 compressed URL lists are smaller in size that the tables of the hash chain approach in [15]. CRC32-based URL lists are also easier to update and simplify switching decisions compared to the methods in [7] and [15].

Table 3. CPU time to generate a compressed URL list

Access list name	Time for URL list	Time per URL
www.peak.org	<10 msec	-
SDMA	<10	-
UVA	40	0.8730 µsec
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

4.4 Evaluation of signature vs. full URL look-up

In this experiment we evaluated the CPU time required to look-up all the entries from an access list in the associated URL list. The URL list is stored as a simple chained hash table with either 1) full URL as the key, or 2) compressed (CRC32) URL 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). Figure 5 shows the total time to look-up 50,000 entries from a URL list based on the LNAR log H. 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).

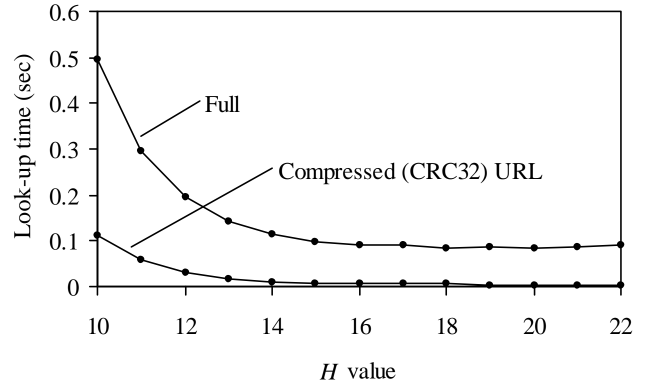

Figure 5. Effects of hash table size on look-up time

E. Evaluation of TCP connection overhead

We evaluate the cost of TCP connection overhead by implementing a simple client/server sockets application. The server listens, accepts, and closes. The client connects and closes. To eliminate network round-trip time, both the server and client are run on the same machine (where the IP packets are wrapped by the adapter device driver). On an 866-Mhz Dell Dimension PentiumIII with Windows2000, the connection rate was found to be about 1200 per second (with 100% CPU utilization). With both the client and server running on the same machine, this measurement is somewhat pessimistic. However, it shows that the CPU requirements for connection overhead are greater than the requirements for URL table look-up. This agrees with findings in [2].

For single CPU routers (e.g., the one-armed router), emphasis should be placed on streamlining TCP connection overhead. For specialized routers (where connections are handled separately from look-up), the benefits from faster URL look-ups are still needed.

5.Summary and Future Work

URL routers can improve the performance of existing cache structure and distributed server CDNs by offloading redirection and forwarding from the content sources. We have shown how URL signatures can be used to improve URL routing. URL signatures based on CRC32 can reduce or compress the size of URL lists and speed-up look-up of URLs in a hash table. A compressed URL list requires less network bandwidth to transfer and less memory for storage in the URL router. For CRC32, the number of collisions in a typical access list was found to be negligible. Misrouted requests to caches will “fix themselves” when the cache retrieves the requested object from the origin server. Misrouted requests will also

occur as a result of normal expiration of content within caches. We also demonstrated methods of generating a CRC32 for subfields within a packet (such as the URL within an HTTP request).

We also intend to explore other applications of signatures to switching. This may include applications to IPv6 routing. Future work includes investigation of table update issues. We also intend to continue the work in [10] on inherent instabilities in server selection with delayed load information to gain insights on the table update frequencies needed between URL routers.

Acknowledgements

The authors thank Allen Roginsky from IBM Corporation for his valuable insights on CRC properties.

References

[1] D. Andresen, T. Yang, V. Holmedahl, O. H. Ibarra, "SWEB Towards a Scalable World Wide Web Server on Multicomputers," *Proceedings of the 10th IEEE International Symposium on Parallel Processing (IPPS'96)*, pp. 850-856. April 1996.

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

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

[4] M. Crawford , "Non-Terminal DNS Name Redirection," RFC 2672, August 1999.

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

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

[7] 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, October 1998.

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

[9] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier, "Cluster-Based Scalable Network Services," *Proceedings of the 16th Symposium on Operating Systems Principles*, pp. 78-91, October 1997

[10] 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.

[11] Z. Genova and K. Christensen, "Efficient Summarization of URLs using CRC32 for Implementing URL Switching," under preparation, 2001.

[12] 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.

[13] D. A. Maltz and P. Bhagwat, "TCP Splicing for Application Layer Proxy Performance," *IBM Research Report RC 21139*, March 1998.

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

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

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

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

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

[19] Personal conversations with Azer Bestavros, April 2001.

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

[21] 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.

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

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

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

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

Appendix A - Handling URL Prefixes

For caches it is unlikely that an entire directory structure from an origin server will be reproduced. However, a distributed server may contain entire directories of thousands of files from an origin server. For these cases, longest prefix matching can allow the URL routing table to contain only prefix signatures. Prefix signatures are CRC32 for the URL components separated by slashes (e.g., the hostname, directory, and filename are stored as three CRC32 signatures). Look-up in the hash table is done in backwards order starting from the full URL signature. The first key match in the hash table is the longest matching prefix. Multiple look-ups may be required per URL look-up.

Appendix B - Generating CRC32 Signatures

This appendix describes how 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. It is assumed that the current value of the CRC32 shift register can be sampled at byte boundaries.

We will use the CRC properties described and proved in [21]. Let P be a CRC generator polynomial. For any polynomials (bit sequences) A_i, $i = 1, \ldots, m$ we have the following properties,

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

and,

$$\text{Rem}\left(\frac{\prod_{i=1}^{m} A_i}{P}\right) = \text{Rem}\left(\frac{\prod_{i=1}^{m} R_{A_i}}{P}\right) \quad (2b)$$

where,

$$R_{A_i} = \text{Rem}\left(\frac{A_i}{P}\right). \quad (3b)$$

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. We store in a look-up table the remainders corresponding to all possible lengths of A_2, that is, $R_M = \text{Rem}(2^M/P)$. We have $R_{A_0} = \text{Rem}(A_0/P)$ and $R_{A_1} = \text{Rem}(A_1/P)$ returned from CRC32 circuit (e.g., from the network adapter). We wish to find $R_{A_2} = \text{Rem}(A_2/P)$ to have the CRC32 for the subfield A_2. We solve for R_{A_2} as follows. Let A_3 be the bit sequence A_0 shifted left M bits (i.e., multiplied by 2^M). Getting R_M from a look-up table, from (2b) we obtain

$$R_{A_3} = \text{Rem}\left(\frac{A_0 \cdot 2^M}{P}\right) = \text{Rem}\left(\frac{R_{A_0} \cdot R_M}{P}\right). \quad (4b)$$

Now, we can finally perform a subtraction, since A_3 is of same length as A_1. From (1b) we obtain

$$R_{A_2} = \text{Rem}\left(\frac{A_3 - A_1}{P}\right) = \text{Rem}\left(\frac{R_{A_3} - R_{A_1}}{P}\right). \quad (5b)$$

Eq. (5b) follows from modulo-2 addition and subtraction being the same (i.e., an XOR operation). Solving for Eq. (4b) requires a 32-bit multiplication. A 32-bit multiplication can be implemented in software on a 32-bit processor using the algorithm of Fig. 1b.

```
; Inputs m1 and n1, output 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 1b. 32-bit multiply algorithm (from [21])