A Protocol-Independent Technique for Eliminating Redundant Network Traffic

Neil T. Spring and David Wetherall*
Computer Science and Engineering, 352350
University of Washington
Seattle, WA 98195-2350

ABSTRACT

We present a technique for identifying repetitive information transfers and use it to analyze the redundancy of network traffic. Our insight is that dynamic content, streaming media and other traffic that is not caught by today's Web caches is nonetheless likely to derive from similar information. We have therefore adapted similarity detection techniques to the problem of designing a system to eliminate redundant transfers. We identify repeated byte ranges between packets to avoid retransmitting the redundant data.

We find a high level of redundancy and are able to detect repetition that Web proxy caches are not. In our traces, after Web proxy caching has been applied, an additional 39% of the original volume of Web traffic is found to be redundant. Moreover, because our technique makes no assumptions about HTTP protocol syntax or caching semantics, it provides immediate benefits for other types of content, such as streaming media, FTP traffic, news and mail.

1. INTRODUCTION

Much work has focused on the problem of improving Web performance by reducing download times and bandwidth requirements. However, despite the adoption of browser and proxy caching and use of end-to-end compression standards such as JPEG, Web performance still lags user demands. Moreover, proxy caching appears limited in its ability to further improve Web performance. A recent study [19] predicts that proxy caches with Squid [1] cacheability rules can eliminate at most 45% of the traffic – a valuable start, but not a complete solution.

In this paper, we analyze the redundancy present in network traffic with a view to eliminating it as a means of improving Web performance. To the naive user, the behavior of the Web can be quite strange: it can take a long time to download what appears to be the same page, or one with similar content. Such redundant, uncached content has a number of possible origins. It may be:

- dynamically generated or personalized;
- mirrored on a different server ;
- named by a different URL;
- delivered using a new or unsupported protocol;
- updated static content;
- access counted for advertising revenue.

Regardless of origin, these transfers represent a source of inefficiency that we aim to remove. One possible approach is to improve caching protocols to deal well with these situations. However, caching entire documents by name is fundamentally too coarse grained to suppress the redundant content we just described. Documents may need to be transferred in their entirety because they fit into any of the above categories, even if they are quite similar to data that has already been transferred. In addition, application-level caching must adapt as new protocols carrying new types of information such as streaming media emerge and become popular. Explicit cache support for each protocol must be added and deployed, and in the interim, mismatches between usage and cache support will result in unnecessary transfers.

Instead, we explore in this paper a new, protocol-independent technique for identifying redundant information that does not suffer from these disadvantages. The technique is an adaptation of algorithms first proposed by Manber [11] to identify similar files in a file system. Because it identifies similarity between different documents, it generalizes other content-based schemes of which we are aware, such as delta-coding [13] and duplication suppression [12]. It is also efficient: our workstation-based prototype runs at data rates of approximately T3 (45 Mbps).

We have used our redundancy suppression technique to analyze network traffic and find that there is a high level of repetition in the information being transferred to and from our trace site. The potential for byte savings by exploiting this redundancy is the focus of this paper. By comparing packet contents, we find 30% of all incoming traffic and 60% of all outgoing traffic is redundant by our measure. Of the

*email: {nspring,djw}@cs.washington.edu

incoming Web traffic, 14% is cacheable by a proxy cache that uses Squid rules, and a further 41% is then identified as redundant. We find nearly 25% redundancy in web traffic proxy caching does not, including 40% redundancy in documents Web proxy caches are prohibited from caching. This suggests that redundancy suppression and Web proxy caching would work well in combination.

The rest of the paper is organized as follows. We describe our technique for finding redundancy in Section 2. In Section 3, we present the architecture of a caching scheme that would exploit this redundancy to reduce bandwidth use over a bandwidth constrained link. We describe our implementation, including the choice of algorithm parameters, in Section 4. We present an analysis of the redundancy in network traces, paying particular attention to Web traffic, in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2. FINDING REPETITION WITH FINGERPRINTS

Our goal is to process a stream of packets and, for each input packet, quickly identify regions that are repeated from earlier packets. To do this, we adapt a technique developed by Manber for finding similar files in a large file system [11] and applied by Broder to detect similar Web documents [3]. A set of representative fingerprints are computed for each object. Fingerprints are integers generated by a one-way function applied to a set of bytes. Since good fingerprint algorithms generate well-distributed fingerprints, each fingerprint is not only compact but also unique with high probability. Manber and Broder then compare the representative fingerprints of different files to estimate their similarity.

In contrast, we use fingerprints as pointers into the data stream to find regions of repeated content. We store representative fingerprints in an index that maps a fingerprint to the region it describes in a cached packet payload. These fingerprints are "anchors" that are used as hints for finding larger regions of repeated content, both before and after the fingerprinted region.

2.1 Computing Representative Fingerprints

The representative fingerprints for a packet are generated by computing a Rabin fingerprint [14] for every β length substring of the packet, and selecting a deterministic subset of these fingerprints.

A Rabin fingerprint for a sequence of bytes $t_1, t_2, t_3, \ldots t_\beta$, of length β is given by the following expression, where p and M are constant integers:

$$RF(t_1, t_2, t_3 \ldots t_\beta) = (t_1 p^\beta + t_2 p^{\beta-1} \ldots + t_{\beta-1} p + t_\beta) \mod M$$

The form of this expression makes fingerprinting each β length substring $\{\{t_1, t_2, \ldots t_\beta\}, \{t_2, t_3, \ldots t_{\beta+1}\}, \textit{etc.}\}$ computationally efficient. If we compute the fingerprints of a window of size β over the packet from beginning to end, then at each step, the next fingerprint can be defined in terms of the previous one:

$$RF(t_{i+1} \ldots t_{\beta+i}) = (RF(t_i \ldots t_{\beta+i-1}) - t_i \times p^\beta) \times p + t_{\beta+i} \mod M$$

For fast execution, $t_i \times p^\beta$ is precomputed and stored in a table. Since β and p are constant, this table has 256 entries. Rather than generate a new fingerprint from scratch, advancing the fingerprint in this manner requires a subtraction, a multiplication, an addition, and a mask (by $M-1$ to perform the $\mod M$ operation, where M is a power of two).

It is worth noting that MD5 [15], SHA [2], and other secure fingerprint algorithms do not allow this decomposition for incremental computation. They are not efficient for our purpose.

2.2 Fingerprint Selection

It is impractical to index every fingerprint that is computed: it would require nearly one index entry per byte! Yet we cannot simply select every nth fingerprint and locate shifted content. That is, if two packets are identical except that one has an extra byte inserted at the beginning, no redundancy would be found with the "select every nth fingerprint" strategy.

Manber's insight was to select a fraction of the fingerprints depending on their values, not locations. Selecting a fraction of fingerprint values gives a deterministic sample of content that is not sensitive to location. This means that the same content that is packetized in different ways and interspersed with other data can be recognized as repetitive. Because the fingerprints are random and uniformly distributed, any fraction provides probabilistically good coverage. Selecting a fraction is easily accomplished in practice by selecting those binary fingerprints that end in a specified number of zeros; we use γ to denote the number of zeros in our experiments.

2.3 Algorithm

Our algorithm for finding repeated content is run for every packet of a (possibly infinite) input stream. A cache is used to hold the most recent packets, and it is this cache against which the input packet is checked for redundancy. The cache is indexed by the representative fingerprints of the packets that it holds.

For every packet, the algorithm first generates the representative set of fingerprints. Each fingerprint in this set is checked against the index of the cache. If it is found, then a packet in the cache has the same content as the input packet in the regions that correspond to the fingerprint. This region is compared in the input and cached packets to verify that there has not been a collision in the fingerprint name space. We use fingerprints that are sufficiently large that we observed no collisions in practice. Then the matching region is expanded, both to the left and to the right, byte-by-byte in each packet, to find the largest matching region. The total repeated content that is found is then simply the union of all largest matching regions. Finally, the packet cache and fingerprint index are updated by inserting the newly processed packet, evicting the oldest packets in FIFO order

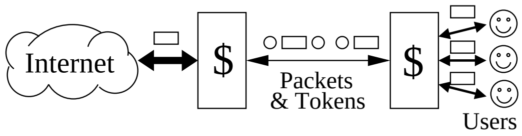

Figure 1: Shared cache architecture. Tokens are passed in place of replicated bytes between the two caches ($) on opposite sides of a bandwidth constrained link. User machines continue to communicate using unencoded packets.

to make room if necessary. If an index fingerprint refers to a packet that is no longer in the cache, it is released.

There are two further details. First, we always select the fingerprint at the beginning of every packet (or over the whole packet if it is shorter than our fingerprint window) to ensure that every packet is represented in the cache at least once. Second, fingerprint generation and content matching are actually interleaved, rather than sequential as our exposition implies. This does not change the algorithm conceptually, but results in a more efficient implementation.

3. SHARED CACHE ARCHITECTURE

We envision that our redundancy suppression technique can be used to improve Web and other protocol performance with the architecture shown in Figure 1. Caches running our algorithm are placed at both ends of a bandwidth-constrained channel. This channel could be an access link or wireless link, or an end-to-end network path between server and proxy or server and client. Each cache converts repeated strings into tokens and passes these encoded, smaller packets to the other end of the channel, where the original packet is reconstructed. This trades memory and computation for bandwidth savings. We emphasize that we have not implemented this overall architecture. The focus of this paper is the redundancy that is present in network traffic and potential bandwidth savings, which we characterize by analysis; we describe an architecture to provide the appropriate context for discussion.

One key problem that we have not explored but which must be addressed in a complete implementation is the synchronization mechanisms that keep the contents of both caches consistent. If the caches lose consistency because of packet losses, then the tokens cannot be decoded at the far end of the channel. Fingerprints are likely to be of value here. This is because, assuming there are no collisions, the loss of consistency can be detected when an unrecognized fingerprint token is received. This can then be repaired by requesting the missing packet from upstream.

There are two other aspects of our architecture worth noting. First, it can improve the performance of non-Web protocols. This is because the matching technique is completely protocol independent. Second, it complements rather than replaces traditional Web caching. This is because by making use of application level semantics Web caches can sometimes eliminate the need to contact the remote server, thereby directly cutting latency as well as saving bandwidth. Without an analogous understanding, our system cannot dispense with a Web transfer entirely. Rather, it is comparable to a sophisticated GET-IF-MODIFIED mechanism across the channel that returns only the parts of a packet or document that have not already been sent across the channel.

4. IMPLEMENTATION

We describe the implementation of our redundancy analysis engine in terms of its design, selection of operational parameters, and performance.

4.1 Design

We implemented our analysis engine as a user-level process running on a PC that processes a trace file and outputs statistics on its redundancy. The design is a straightforward realization of the algorithm described in Section 2 with the following features:

- We chose M, the base of the modular arithmetic performed in the fingerprinting algorithm, to be 2^{60}. This is sufficiently large to ensure that there are no fingerprint collisions for the range of memory sizes that we used, but also implies lower performance for fingerprint calculations on 32-bit machines. We chose p, the factor used in computing the Rabin fingerprint, to be 1,048,583, a large prime.
- We use a simple first-in first-out policy across all packets to manage the packet store. Clever implementations could improve on the redundancy we detect with an admission policy that treats different types of data differently, an eviction policy that holds referenced packets in the cache longer, or by screening to avoid multiple copies of the same content.
- Our index maps each fingerprint to the most recent packet in the cache, for simplicity and performance.
- We strip packet headers including UDP/TCP before searching for redundancy. This is not required for our analysis, but improves efficiency, because headers are typically not repetitive. Effective schemes exist for compressing TCP headers [10].
- As part of computing the number of redundant bytes, we charge each match region a small penalty that is intended to represent the space needed to encode it for transmission. We felt this penalty important because, if it were not present, we could ultimately detect 100% redundancy by observing that every short combination of bytes is repeated! The penalty we use is 12 bytes, which is ample to encode a fingerprint plus a description of the matched range. This description consists of the offset of the fingerprint in the packet, with a count of redundant bytes before and after, encoded as three, 11-bit integers, sufficient for Internet datagrams. In practice, this penalty has a small effect (a few percent overhead) because the average match region is hundreds of bytes long.

4.2 Algorithm Parameters γ and β

The first task we faced when the implementation was complete was to determine the remaining algorithm parameters. Recall that γ determines the fraction (an average of 1 out

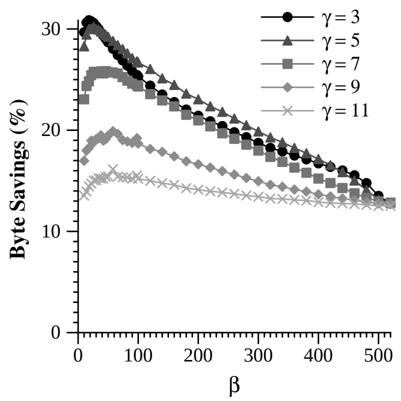

Figure 2: Effect of γ and β on the redundancy that is found. We used $\gamma = 5$ bits and $\beta = 64$ bytes for subsequent experiments.

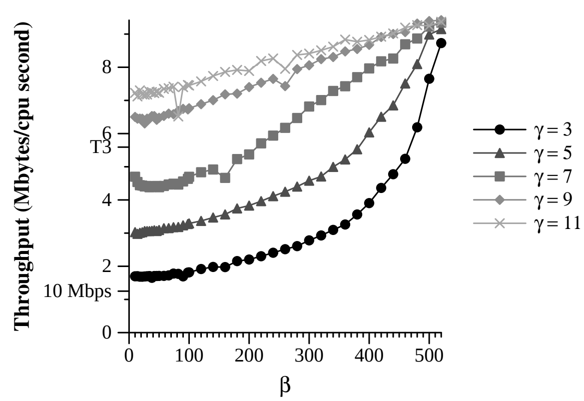

Figure 3: Throughput as a function of β at different values of γ. Cache size was fixed at 128 megabytes.

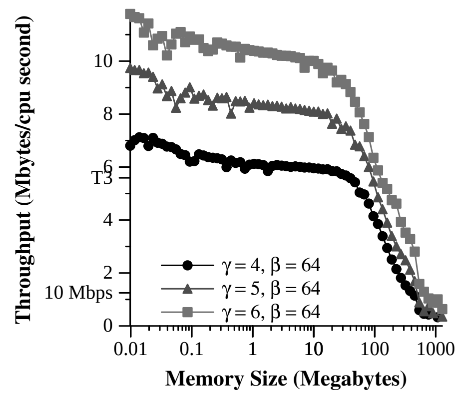

Figure 4: Throughput as a function of cache size.

of every 2^γ) of fingerprints that are selected, and β determines the minimum width of match regions. We were most concerned about the choice of β. If β is too large, only large regions are matched, which increases the average "quality" of matches but decreases the potential byte savings that is detected. If β is too small, the average "quality" of matches is sacrificed, since shorter matches may be more recent than better, longer matches. There is also a tradeoff with γ in terms of how well (frequently) each packet is sampled, which can increase the likelihood of finding a match for a given packet but cut into the memory available for storing all packets.

In practice, these tradeoffs were not problematic. Figure 2 shows the amount of redundancy that is found in our trace data for different values of γ and β. (The trace data is described in Section 5.) Small β and γ are most effective, and we limit the values we choose only due to the performance considerations discussed in the next subsection. We chose $\beta = 64$ bytes and $\gamma = 5$ bits for subsequent experiments. This setting results in memory consumption split roughly 40% index / 60% packet cache, and finds close to the maximum redundancy. The proportion of index may seem large; indexing must be comprehensive in order to be useful.

4.3 Performance

To gauge the approximate performance we could expect in practice, we measured the throughput of our implementation over a range of parameter settings. This was done on a dual processor Pentium III-550 with 1GB of memory, running Linux 2.2.9. The second processor should have negligible effect, since only one processor is used by the analysis. Throughput was measured as the number of packet payload bytes of trace data processed per second of CPU accounted time, as reported by the Unix `time` utility.

Figure 3 shows throughput as a function of β and γ with a cache of 128 MB. The computational load increases as β and γ decrease: more fingerprints are computed and indexed as representative. It is also clear that γ should be as large as possible if performance is an issue; as γ falls, the size of the index grows exponentially, and this degrades performance by both interrupting the fingerprinting process often and by reducing the memory locality of operations.

Figure 4 shows throughput as a function of memory size. For $\beta = 64$ bytes and $\gamma = 5$ bits, the parameters we selected for experiments, throughput is between 10Mbps and 45Mbps (T3) rates. Throughput falls with memory size, especially for sizes above 10 MB, presumably due to memory cache behavior. We expect little locality of access index because of the randomizing effect of the fingerprints.

These results suggest that reasonable performance can be obtained with modest effort. In a system with sufficient network interface to memory performance, speeds of at least T3 should be readily achievable, and likely much better with tuned or in-kernel implementations.

5. TRAFFIC ANALYSIS

In this section, we present the results of a trace-based analysis of the redundancy of network traffic. We also compare the redundancy found by our protocol independent technique with that of alternative approaches, such as Web proxy caching and packet compression. Overall, we find that there is a high level of redundancy, especially in Web traffic (28%), and that this redundancy is not adequately exploited by alternative approaches.

5.1 Trace Data

The trace data we analyze comes from a corporate research environment consisting of roughly 3000 users and a handful of web servers. These traces include all packets exchanged between the site and the rest of the Internet across the network connecting the border router and firewall gateways, a region commonly called the DMZ. To the best of our knowledge, Web proxy caches are not used within the site.

We use a mix of online and offline analysis. We analyze data online when possible in order to process a large volume of traffic. However, online analysis is limited by the available computation and memory of the monitoring system. We user offline analysis over a shorter volume of traffic when there are insufficient resources for online analysis.

We analyze six traces for offline analysis. The traces were taken Tuesday, November 16th through Thursday, November 18th, 1999, at 10am and 2pm Eastern Standard Time, each for about an hour. Our intent was to capture a representative sample despite variation by time of day and day of the week. The trace machine did not lose any packets during the capture. In total, we analyze 30 million packets and 10 GB of data. This is the limit of our current analysis capability because we must save and process entire packets, not just their headers.

5.2 Amount of Redundancy

The first question we considered was the amount of repetitive data embedded in the traces. This is a function of the size of our cache. Recall that the cache memory is divided into packet storage and fingerprint index portions. We include both portions when describing the required memory size. Also, recall that when we calculate the amount of redundancy we deduct for every match region a small penalty (12 bytes) needed to describe it in a real system; this prevents us from ultimately claiming 100% redundancy because every byte has been seen previously.

We find that more than a third of the traffic is redundant. The results for incoming traffic and outgoing traffic differ, and are shown in Figure 5. For each class of traffic, significant redundancy (at least 10%) is found with relatively little memory (less than 1 MB). Redundancy then increases better than logarithmically with memory size up through 100 MB. Outgoing traffic is almost twice as redundant as incoming traffic, levelling off around 50% with 200 MB of memory. This corresponds to the "working set" of the site's Web servers. Incoming traffic is more diverse, rising to 30% redundancy with 400 MB of memory. It also appears that more redundancy would be observed in systems where more memory was available, e.g., with a disk-based packet store. Based on these results, we use a memory size of 50 MB for on line and 100 MB for off line analyses in this paper.

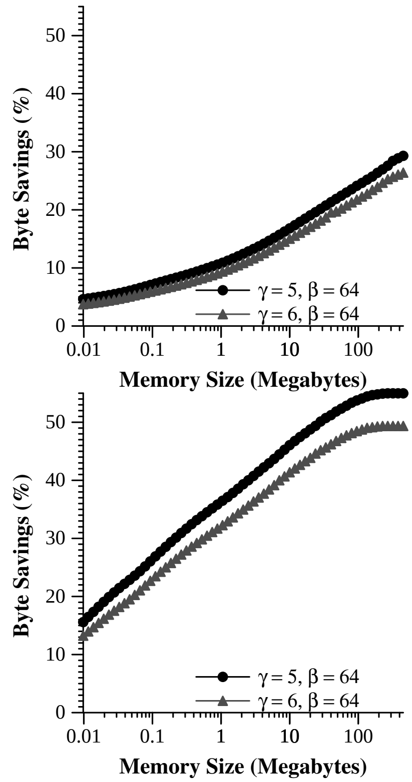

Figure 5: Redundancy found as a function of total memory allocated. Incoming traffic (top) and Outgoing traffic (bottom) are separated.

5.3 Redundancy by Protocol

To narrow the source and type of redundancy, we classified traffic into different protocols and calculated the amount of redundancy for each protocol. This was done by using well-known ports and matching against both the source and destination port. The results are shown in Table 1. We consider only incoming traffic, and use a 200 MB cache.

The bulk of the traffic is Web-based, agreeing with common perception and [5, 6, 17]. We exclude cacheable web responses, and find that this traffic is highly repetitive (30% redundant). After Web traffic, a significant fraction of the remaining traffic is streaming media, delivered using protocols developed by Real Networks and Microsoft. This traffic is mildly redundant (7%-26%). Many of the less-significant traffic contributors are also surprisingly redundant. In particular, Telnet and Lotus Notes are 36% and 18% redundant, respectively. Finally, we note that a significant amount of traffic (15%) and redundancy (10%), categorized as "Other," is carried on high-numbered ports that we have not yet been able to associate with well-known services.

5.4 Peak Traffic Periods

How effective would a redundancy suppression engine be at peak usage times? Is traffic likely to be more or less redundant when there are many clients? How fast would a link, operating at capacity, appear to be if enhanced by redundancy suppression? To answer these related questions, we output the number of redundant bytes and the total number of payload bytes every two minutes for two months, from February 25 to April 25, 2000. The cache size was only 50 million bytes, since the machine had only 128 MB during this analysis.

Figure 6 shows the fraction of traffic that is redundant as a function of the traffic rate. We draw two conclusions from

Media Type	Protocol (Ports)	Traffic by Byte	Byte Savings
Web service	HTTP (80, 8000, 8080)	64.3%	30%
Real streaming media	RTSP (554, 7070)	7.3%	7%
Music transfer	Napster (6688, 6699)	2.7%	6%
Lotus Notes	Lotus (1352)	2.4%	18%
Secure Web	HTTPS (443)	2.3%	4%
File transfer	FTP-data (20)	1.9%	4%
Usenet news	NNTP (119)	0.9%	7%
Name service	DNS (53)	0.7%	15%
Microsoft media	ASF (1755)	0.6%	26%
AOL	AOL (5190)	0.6%	16%
Name service	SMTP (25)	0.5%	20%
Mail receipt	POP (109, 110)	0.3%	28%
Remote terminal	Telnet (23)	0.1%	34%
Unknown	Other	15.3%	10%

Table 1: Redundant incoming traffic by protocol, sorted in decreasing order of traffic contribution. The volume of incoming traffic, excluding what would be cached by a proxy cache, was 97.5 bytes. Web service includes both incoming requests and uncached incoming responses.

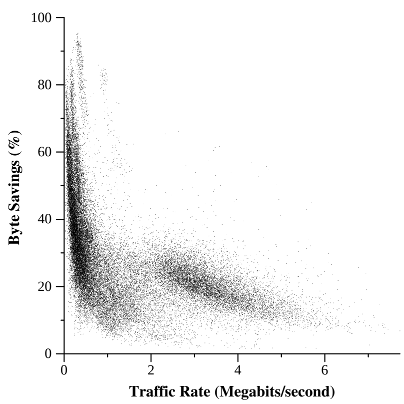

Figure 6: How does the percentage of redundant traffic vary with link utilization?

these results. First, the redundant fraction over two minute periods can extend beyond 80%! Since this is at periods of lesser activity, this has less practical use. At off-peak times, fewer clients are likely to be active, so the cache is split across fewer clients. We believe that this implies the performance of the system should scale well for additional clients with additional cache memory. Second, we notice that the redundancy generally stays around 20%, even at relatively high link utilization.

5.5 Combination with Web Caching

To compare our approach with Web caching, we studied Web traffic that would not have been removed by a proxy cache, even had one been deployed. There are a number of reasons why documents may not be cached in addition to cache misses, for example, to protect user privacy or ensure that a page is fresh. These motivations are expressed in the set of cacheability rules used by the Squid [1] proxy cache. The following request characteristics make a document uncacheable:

- **Question**: The object name includes "?".
- **CGI**: The object name includes "cgi-bin" or "htbin".
- **Cache Control**: The HTTP 1.1 request includes a "Cache-control" header, such as "private", "no-cache", and "no-store".
- **Method**: The request is neither GET nor HEAD.
- **Pragma Request**: The request includes "Pragma: no-cache" in its header.
- **Authorization (Auth)**: The request header includes an "Authorization" header for access control.

Older caching proxies would not cache responses to requests that included cookies [4, 7]. Since Squid [1] version 2 will cache such HTTP 1.1 responses, we do not consider requests that include cookies to be uncacheable.

In addition, the *response* may have uncacheable properties:

- **Cache Control**: The HTTP 1.1 response header includes a "Cache-control" header that prohibits caching.
- **Pragma Response**: The response includes "Pragma: no-cache" in its header.
- **Set Cookie**: The response attempts to set a cookie for future use by the client.
- **Response Status**: The response status code is one which prohibits caching, such as 307 Temporary Redirect or 401 Unauthorized [8].
- **Push**: The "content-type" is one which suggests that the server may continuously stream data to the client.

For the analysis we present here, each incoming HTTP response was classified as cached, cacheable but not cached, or uncacheable. Packets from responses that were not cached were passed to the redundancy suppression engine. The

Document Type	Traffic Volume (% total bytes)	Redundancy (% of category)
Cached Web	12.6%	n/a
Uncached Web	87.4%	23.3%
Cacheable	63.4%	15.6%
Uncacheable	24.0%	41.4%

Table 2: Summary of Web document cacheability. Documents that were not cached are either cacheable, consisting of cache misses, or uncacheable, consisting of responses that can not be cached. The total Web traffic observed consisted of 108.5 GB in 11,024,951 requests.

Reason Uncacheable	Traffic Volume (% total bytes)	Redundant (% of category)
Question	10.7%	52.2%
Method	5.0%	17.4%
CGI	4.2%	53.7%
Cache Control	3.4%	32.0%
Pragma Response	2.3%	76.0%
Pragma Request	1.6%	39.6%
Set Cookie	1.4%	39.7%
Authorization	0.8%	30.6%
Response Status	0.4%	36.3%
Push	0.2%	3.3%

Table 3: Summary of the redundancy of uncacheable documents by reason for their uncacheability. The total Web traffic observed consisted of 108.5 GB in 11,024,951 requests.

fingerprints of cacheable object names were inserted into a 64,000 entry, list, held in least recently accessed order. Since the average size of a Web document is 8KB [20], this roughly simulates a 512 MB Web cache. This is a smaller cache than we would like to simulate, but the limit of our analysis system. Documents that were uncacheable were tagged with their uncacheable attributes. Some may be tagged with more than one; often both "cgi-bin" and "?" appear in the same request. Our analysis ran with a 200 MB redundancy suppression cache from May 19 to May 26, 2000. A summary of the byte savings found in Cached, Uncacheable, and Cacheable (but missed in the proxy cache) documents is presented in Table 2.

Table 3 presents the byte savings we found in each class of uncacheable document, ordered by the volume of traffic each one represents. Uncacheable documents represent only one quarter of the bytes transferred. A tremendous amount of redundancy is found in those responses that include the "Pragma: no-cache" header, while very little redundancy is found in those responses for which authorization was necessary.

HTTP requests and responses were identified by examining packet headers without consideration for source or destination port numbers, so the volume of traffic does not agree with that in Table 1.

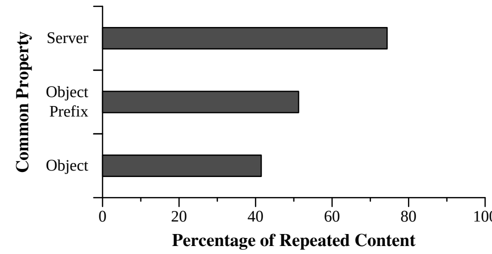

Figure 7: When a redundant match is found, what characteristics do the new and old packets share? The total volume of repeated content was 22.7 GB.

5.6 Locality of Redundancy

In this section we attempt to determine whether an end-to-end solution would be appropriate for suppressing redundancy. That is, is there significant redundancy between streams originating from different servers? If so, a solution within the network, such as the paired caches we describe in Section 3 would be appropriate to protect a slow link. If not, then such a solution within the network would have little benefit over an end-to-end solution.

To answer this question, for each redundant string of bytes we find, we consider whether various attributes of both the newly arrived packet and the stored packet match, when both are from HTTP responses.

- **Server** If the source IP addresses match completely, the redundancy is from the same server, and could be addressed using a server to proxy system.
- **Object Prefix** If it is known which HTTP object was requested, and the text of the requested objects match up until the "?" (if any), we consider the "Object Prefix" to match. This is for comparison with various approaches (e.g. [9]) that leverage matching object prefixes to find redundancy.
- **Object** If the above holds, and the rest of the object name matches, we consider the redundancy to be local to an object. This includes both repeated transfers of the same object, and similarity found within a single transfer of an object.

The scope of locality is summarized in Figure 7. Of the total redundancy found on-line in Web traffic, represented by the x-axis, each bar represents the number of bytes found by matching packets from similar sources. Figure 7 suggests that an end-to-end solution would capture the bulk of the redundancy. That is, redundant traffic is generally (78% of the time) from the same server, so the benefit of looking for redundancy in another server's traffic is small.

Figure 7 also demonstrates that solutions that seek similarity only between transfers of documents with the same name miss out on a significant source of additional redundancy. A server to proxy, content-based solution is likely to detect significant redundancy that a name-based cache can not.

Caching / Compression Algorithm	Byte Savings
Web Proxy	14%
zlib Packet Compression	16%
Web Proxy and zlib Compression	28%
Redundancy Suppression	54%
Web Proxy and Redundancy Suppression	56%

Table 4: Average byte savings over the 6 off-line traces for different approaches . The cache size was 100 million bytes.

Since our clients are anonymous, we do not correlate the redundancy by destination, so we do not compare a server to proxy system with a server to client system. However, we expect that there is less locality in client requests than in server responses, and that a proxied system would be advantageous.

5.7 Comparison with Packet Compression

To compare the value of our technique with more traditional compression algorithms, we use a packet compression tool instead of our redundancy analysis. Lacking a standard packet level scheme that is in widespread use, we elected to individually compress each packet payload with the zlib compression library, the same library used by the gzip utility, as a point of reference.

Compression is able to reduce the off-line traces by 15% on average, less than half of our average gain, and works more or less independently of Web caching. It was also our experience that this form of compression was more compute intensive than our technique.

Moreover, our earlier experiments demonstrate that compression taps a different source of redundancy than our algorithm. The gzip implementation searches a history window of only 32KB when looking for repeated sequences to compress. The results in Figure 5 show that our technique finds increasing amounts of redundancy for memory sizes through 500 MB, well beyond the range of gzip.

The combination of zlib with a caching Web proxy on our off-line traces is able to eliminate 28% of the bytes in the trace. We are encouraged by the orthogonality between the Web proxy and zlib. Combining this with the relative orthogonality of zlib compression with packet caching as described in [16], we expect that an integrated solution, using a Web proxy cache to eliminate requests, redundancy suppression to remove long-term repetition, and zlib to compress individual packets, would significantly improve the performance of low bandwidth links.

5.8 Summary

In this section, we have supported the belief that our redundancy suppression technique complements Web proxy caching and traditional content compression.

The overall off-line byte reductions for each approach is shown in Table 4. The Web proxy alone only eliminates 14% of the bytes. That this is so small does not appear to be solely an artifact of the short time scale of the traces, rather that proxy cache hit rates are often cited in terms of documents, instead of bytes [4]. In fact, the Web proxy hit rate by document we observe approaches 25%. Compression yields similar benefits and operates independently of caching. Redundancy suppression provides still greater benefits that are not captured by proxy caching alone.

Given this result, we believe that our technique complements Web caching and traditional packet compression, and taps a distinct source of repetition. URL-based proxy caching has the advantage that it can service requests without incurring the latency of communication with the remote server. Our technique then finds content-based matches that Web caching does not. This information can readily be used to further reduce bandwidth requirements and, by implication, overall latency.

6. RELATED WORK

Much previous work is targeted at reducing Web-related bandwidth requirements. We omit a general discussion of Web caching and data compression for brevity; we compared our technique to both in the previous section. Instead, we discuss two prior approaches that are most similar to our own.

6.1 Delta Encoding

Delta encoding, proposed by Housel and Lindquist [9], and studied by Mogul et al [13], is motivated by the observation that when pages are updated they are often changed in only minor ways. Thus it can be more efficient to transfer only the changes by using a variant of "get-if-modified" that supports versions.

Conceptually, the main difference between the two is that delta encoding is based on differences between versions of a single document, while our technique is able to efficiently identify differences between all documents. Thus we expect our technique to find strictly more repetition. Evidence to support this belief can be seen in Figure 7, where there is significantly more redundancy available from the same server than under the same URL.

6.2 Duplicate Suppression

Duplicate suppression involves identifying and suppressing the transfer of exact duplicates, which are typically identified by fingerprints. Mogul [12] has explored this approach as proposed in [18] at the document level, and Santos [16] has explored it at the packet level.

Again, conceptually, our proposal extends duplicate suppression in a significant dimension: we look for similar, overlapping information rather than identical, duplicated information. This is a harder problem with potentially greater benefits. There is some evidence to support this claim in Figure 2. As the parameter β is increased, only longer regions are considered for matching. The large β values on the right side of the graph thus correspond to whole packet matching – the equivalent of packet level duplicate suppression. It is apparent from the graph that as β is reduced, up to twice as much redundancy is found. We do not have data to compare directly to document level duplicate suppression, but have no reason to believe that the results would be different. Mogul [12] reports that the value of duplicate suppression varies by data type and on average is low.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a new technique for finding redundancy in network traffic. Our technique builds on the work of Manber to detect similar, but not necessarily identical, information transfers. In terms of improving Web performance, it has the potential to exceed the benefits of other approaches such as delta coding and duplicate suppression. This is because the similarity algorithms on which it is based include as a subset both exact matches (duplicate suppression) and differences between versions of the same document (delta coding).

A distinguishing feature of our system is that it is protocol independent. It makes no assumptions about the syntax or semantics of HTTP. This has two distinct advantages. It is able to identify fine-grained sharing, as may be common with dynamically generated or personalized pages, as well as inter-protocol sharing. It does not need to be updated to take advantage of new types of content, such as streaming media, as they emerge or delivery protocols are revised.

We used our technique to characterize the redundancy of network traffic. We found a high degree of repetition in dynamically generated Web documents: roughly 50% in CGI and Question categories. For a class of documents for which the server prohibited caching, this redundancy approaches 76%. In uncached web documents, 22% of the redundancy we found originated from different Web servers. Of the redundancy from the same server, only two thirds is found between objects of the same name. In our protocol analysis, we found byte savings ranging from 4-34% in different protocols, and 10% savings in protocols we were unable to identify. These sources of redundancy are not detected by Web caches and could be exploited to improve performance.

The next step we face is to build a complete system that uses our technique, most likely in conjunction with Web caching. Experience with our implementation so far has convinced us that the basic algorithm is well-suited to online use in the network or Web infrastructure. Certainly it is sufficiently fast to find applications: even our untuned, user-level prototype can run at approximately 45 Mbps on PC hardware. We speculate that, with increasingly large amounts of memory in the future, protocol independent caching may be applied across bandwidth-constrained links routinely, as a well-known technique to trade memory for bandwidth.

8. ACKNOWLEDGEMENTS

We wish to thank Alec Wolman and Maureen Chesire for their help with the HTTP cacheability analysis software from [20]. We thank Srinivasan Seshan and Vern Paxson for their assistance. Udi Manber furnished us with the source for siff as described in [11]. Jeff Mogul kept us from giving him too much credit in Section 6. Gretta Bartels, Stefan Savage, Mike Swift, Tom Anderson, Hank Levy, and the anonymous reviewers provided helpful feedback and greatly improved the quality of this paper. This research was supported in part by DARPA Grant F30602-98-1-0205. Neil Spring was partially supported by a W. Hunter Simpson Fellowship from the ARCS Foundation.

9. REFERENCES

[1] Squid Web proxy cache. `http://www.squid-cache.org/`.

[2] National Institute of Standards and Technology, Specifications for secure hash standard, April 1995. Federal Information Processing Standards Publication 180-1.

[3] A. Broder. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29, March 1998.

[4] R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web proxy caching: The devil is in the details. In *Proceedings of the Workshop on Internet Server Performance*, June 1998.

[5] CAIDA. Traffic workload overview. `http://www.caida.org/Learn/Flow/tcpudp.html`, June 1999.

[6] K. Claffy, G. Miller, and K. Thompson. The nature of the beast: Recent traffic measurements from an Internet backbone. In *Proceedings of INET '98*, July 1998.

[7] A. Feldmann, R. Caceres, F. Douglis, G. Glass, and M. Rabinovich. Performance of web proxy caching in heterogeneous bandwidth environments. In *Proceedings of IEEE INFOCOM'99*, May 1999.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1, June 1999. Networking Working Group Requests for Comment RFC-2616.

[9] B. C. Housel and D. B. Lindquist. Webexpress: A system for optimizing web browsing in a wireless environment. In *Proc. 2nd Annual Intl. Conf. on Mobile Computing and Networking*, pages 108–116, Rye, New York, November 1996. ACM. `http://www.networking.ibm.com/art/artwewp.htm`.

[10] V. Jacobson. Compressing TCP/IP headers for low-speed serial links, February 1990. RFC 1144.

[11] U. Manber. Finding similar files in a large file system. In *Proceedings of USENIX Winter 1994 Technical Conference*, January 1994.

[12] J. C. Mogul. A trace-based analysis of duplicate suppression in HTTP. Technical Report 99/2, Compaq Computer Corporation Western Research Laboratory, November 1999. available from `http://www.research.digital.com/wrl/techreports/abstracts/99.2.html`.

[13] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. Technical Report 97/4, Compaq Computer Corporation, July 1997. available from `http://www.research.digital.com/wrl/techreports/abstracts/97.4.html`.

[14] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Department of Computer Science, Harvard University, 1981.

[15] R. Rivest. The MD5 message-digest algorithm, 1992. Networking Working Group Requests for Comment, MIT Laboratory for Computer Science and RSA Data Security, Inc., RFC-1321.

[16] J. Santos and D. Wetherall. Increasing effective link bandwidth by suppressing replicated data. In *Proceedings of USENIX Annual Technical Conference*, 1998.

[17] K. Thompson, G. J. Miller, and R. Wilder. Wide-area Internet traffic patterns and characteristics. *IEEE Network*, 11(6):10–23, Nov. 1997.

[18] A. van Hoff, J. Giannandrea, M. Hapner, S. Carter, and M. Medin. The HTTP distribution and replication protocol. Technical Report NOTE-DRP, World Wide Web Consortium, August 1997. `http://www.w3.org/TR/NOTE-drp-19970825.html`.

[19] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-based analysis of web-object sharing and caching. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems (USITS '99)*, pages 25–36, October 1999.

[20] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 16–31, December 1999.