Reducing the Network Load of Replicated Data

by

Jonathan R. Santos

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degrees of

Master of Engineering

and

Bachelor of Science in Computer Science and Engineering

at the

Massachusetts Institute of Technology

May 1998 (June 1998)

Author .....

Department of Electrical Engineering and Computer Science

May 20, 1998

Certified by .....

John V. Guttag

Professor and Associate Head, Computer Science and Engineering

Thesis Supervisor

Certified by .....

David J. Wetherall

Research Assistant

Supervisor

Accepted by .....

Arthur C. Smith

Chairman, Department Committee on Graduate Students

Reducing the Network Load of Replicated Data

by

Jonathan R. Santos

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 1998, in partial fulfillment of the
requirements for the degrees of
Master of Engineering
and
Bachelor of Science in Computer Science and Engineering

Abstract

In the Internet today, transfer rates are often limited by the bandwidth of a bottleneck link rather than the computing power available at the ends of the links. To address this problem, we have designed a novel link layer protocol that employs computation at the ends of the link to reduce bandwidth consumption across the link. Our scheme is motivated by the prevalence of repeated transfers of the same information, as typically occurs in the form of HTTP, FTP, and DNS traffic. The protocol complements existing link compression and application-level caching schemes by combining aspects of both. It is able to detect and remove redundancy at the packet level and across large timescales.

We make three contributions in this thesis. First, to motivate our scheme we show by packet trace analysis that there is significant replication of data at the packet level, mainly due to Web traffic. Second, we present an innovative link compression protocol that is well-suited to traffic with such long-range correlation. Third, we demonstrate by experimentation that the availability of inexpensive memory and general-purpose processors in PCs makes our protocol practical and useful at rates exceeding T3 (45 Mbps).

Thesis Supervisor: John V. Guttag
Title: Professor and Associate Head, Computer Science and Engineering

Thesis Supervisor: David J. Wetherall
Title: Research Assistant

Acknowledgments

I would like to show my gratitude and appreciation to the following people whose contributions and support helped me reach this milestone in my life:

- David Wetherall for being a spectacular supervisor, without whom this thesis would be nothing compared to what it is today. I am immeasurably grateful for all his guidance, advice, insightful comments and suggestions, and the many other contributions which are too numerous to list.
- John Guttag for all his encouragement, enthusiasm and support.
- Vanu Bose for stepping in with his help and advice when Dave was busy showing the world how "stupendous" he is.
- Andrew Chiu for letting me know about the opportunity with the Software Devices and Systems group, and for putting in the good word that may have gotten me in.
- Rachel Bredemeier for keeping the cappuccino supplies ready.
- All the members of the Software Devices and Systems, for making my experience with the group fun and worthwhile.
- Michelle Girvan for her understanding, patience, comfort, and support.
- My family for opening up the endless opportunities now available to me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the Internet today, transfer rates are often limited by the bandwidth of bottleneck links rather than the computing power available at the ends of the links. Link-level compression and application-level caching are two approaches used today to alleviate this congestion by trading computational resources for available bandwidth. This thesis presents a new approach that combines the transparency of link-level compression with the bandwidth-saving benefits of caching. Our approach increases the available bandwidth of a link by using general-purpose processors and memory to detect and remove repeated transfers of the same information at the packet level. The abundance of these repeated transfers, as can be found in traffic from information servers, makes this scheme a practical and economical means to improve the effective bandwidth of a heavily-loaded link.

1.1 Motivation

This thesis is motivated by three factors:

- The exponential growth of Internet traffic in recent years has produced a large strain on network resources, creating congestion on low-bandwidth links.
- Much of Internet traffic is from information servers, primarily WWW servers. A common characteristic of these servers is that they often transmit many

instances of the same data across the network.

- The availability of fast and inexpensive memory and general-purpose processors make it practical and economical to apply computational resources within a network.

1.1.1 Congestion on Bottleneck Links

In the past few years, traffic on the Internet has grown exponentially. As of January 1998, approximately 30 million hosts were connected to the Internet, as compared to the 6 million of just three years prior [11]. This increase in the number of hosts, in conjunction with the development of higher-bandwidth applications (e.g., real-time audio and video services), has created a large demand on network resources.

However, many links are limited in their ability to handle the volume of traffic demanded by the clients. For example, access links (modem, ISDN, T1, T3) restrict bandwidth due to cost, while wireless links restrict bandwidth due to properties of the media. Therefore, there exists a need for more efficient utilization of these bottleneck links.

1.1.2 Replication in Traffic

Much of today's traffic consists of data being delivered by information servers such as FTP servers, and more prominently, WWW servers. For example, in a recent five-minute sample of traffic at an OC-3 truck of an Internet backbone, HTTP traffic comprised over 65% of the total traffic volume [8].

A common property of these information servers is that they frequently transmit identical copies of the same information across the same link. There are several reasons why this occurs:

- A particular object may be requested multiple times by the same or different clients.

- Different copies of identical objects passing through a link may originate from different servers or may be referenced by different names. For example, this is often the case when related Web pages contain the same graphical elements, such as logos, icons, background images, and so forth, as well as when information is explicitly "mirrored" on multiple hosts.

- Errors encountered while transferring an object may cause parts of the object or the entire object to be retransmitted.

Previous studies of FTP traffic over the NSFNET backbone [3] and at the main gateway of the University of Colorado [7] indicated that half of the FTP bytes transferred were duplicate transmissions. Since that time, the protocols and traffic patterns have changed with the growth of the Web – it is now HTTP, not FTP, that is dominant. However, the level of redundancy is still high. For instance, a recent analysis of Web traces taken at the Internet gateway for AT&T Labs-Research [6] reports that about half of the references were to previously referenced resources.

1.1.3 Availability of Fast and Inexpensive Hardware

Technological advances in the computer hardware industry have made fast, inexpensive memory and general-purpose processors widely available. For instance, it is now possible to obtain a PentiumII-400 MHz machine with 256 MB of RAM and support for high-speed networking for under $4000. These machines are capable of processing network packets at rates of 100 Mbps, and thus could be useful for manipulating packets within a network.

1.2 Suppression of Replicated Data

The need for more efficient use of network links implies that it would be worthwhile to reclaim the bandwidth consumed by repeated transfers of the same information. For this reason, we have designed a scheme that trades computational resources at the ends of a link in return for increased effective bandwidth. This scheme takes

advantage of the availability of inexpensive memory and general-purpose processors to provide an economic means of effectively purchasing additional bandwidth. Today, it costs $2500 per month to purchase a T1 (1.5 Mbps) link. Our scheme can yield a greater increase in effective bandwidth at a one time cost of two PC's.

Our scheme suppresses transmissions of these replicated data at the link level. This is accomplished by dynamically building a dictionary of the data at both ends of the link. Then, rather than sending replicated data across the link, we send a short dictionary token, which is used to lookup the associated data at the other end of the link.

Our scheme has several interesting properties:

- It is independent of the data format of the packet contents and therefore provides benefits even when application objects have been previously compressed.

- It utilizes a correlation between packets that is not available at individual clients and servers and is not found by existing link compression schemes.

- It provides the bandwidth reduction benefits of caching in a transparent manner. There is no need to modify the end applications or protocols to benefit from this scheme. Further, there is no risk of stale information or loss of endpoint control.

- It constructs names at the link level and does not depend on higher-level protocol names or details. For example, our scheme will suppress replicated data even if it is identified by different URLs or if it is dynamically generated.

Furthermore, this scheme is not intended to compete with existing approaches for dealing with the growth of network traffic, but to complement them. For example, Web caching works at the application level and is limited by high-level names and details, thus it can miss some of the redundancy that occurs at the packet level. On the other hand, Web caching also provides other benefits that our scheme does not. More specifically, Web caches reduce client latency, distribute server load, and

make more efficient use of wide area network resources, rather than improving the bandwidth utilization of a single link.

In order for this scheme to be effective, several technical challenges must be overcome. First, the scheme must be able to identify significant amounts of replication at the packet level, limited by the amount of memory available in a PC. Also, it must quickly create dictionary tokens that can be transmitted efficiently. Finally, the scheme must be made robust with respect to the failures that occur in practice.

1.3 Summary of Major Contributions

The major contributions of this thesis involve the replication in network traffic at the packet level. We summarize our contributions as follows:

- **New Traffic Analysis** We show by packet trace analysis that there is a significant replication of data at the packet level. We characterize the packet-level replication in terms of amount of replication present, the type of data that is likely to be replicated, and the temporal distribution of the data. To our knowledge, there have been no previous studies on replication at the packet level.

- **Design of a Packet-Level Caching and Compression Scheme:** We present a novel link layer caching and compression system which exploits packet-level replication to reduce bandwidth consumption.

- **Economical System Implementation:** We demonstrate through experimentation that our system can be implemented on inexpensive memory and general-purpose processors to provide significant bandwidth savings at rates exceeding T3 (45 Mbps).

1.4 Organization

In the next Chapter, we review the existing approaches used to improve link utilization. In particular, we discuss compression and application-level caching strategies, and explain how our system combines aspects of each approach.

We then investigate the practical potential for improving the effective bandwidth of a link by removing replicated data from the traffic stream. Chapter 3 presents a trace-driven analysis of the replication present in traffic traveling on a link between our site and the rest of the Internet.

Based on these results, we designed an innovative protocol that uses a cache to detect and remove redundancy at the packet level. We describe, in detail, the design of our system architecture in Chapter 4. Chapter 5 evaluates the performance of our system implemented on commodity PC's. Finally, we conclude in Chapter 6 with observations and suggestions for further work.

Chapter 2

Related Work

The high-level goal of this thesis is to design a system which makes more efficient use of the limited bandwidth provided by network links. Two approaches are commonly used today to help achieve this goal: compression and caching. In this chapter, we will review these two areas of related work and show how our scheme combines aspects of each approach.

2.1 Link-Level Compression

When faced with limited bandwidth, higher effective throughput can be obtained by compressing packets (either individually or as streams) as they enter one side of a bottleneck link and decompressing them at the other end. These compression techniques achieve this goal by reducing the number of bytes required to represent a packet as it is transferred over the link. We discuss two types of link-level compression: data compression and header compression.

2.1.1 Data Compression

Some link protocols such as PPP make provisions for the negotiation of a compression scheme between the ends of the link [15, 13]. These data compression schemes typically represent data using a variant of Lempel-Ziv [19] or other lossless encod-

ing. They take as input a packet or short sequence of packets and remove redundant sequences of bytes.

A major benefit of using this type of compression to increase bandwidth is that it is transparent to higher-level protocols and applications because it occurs entirely at the network level. That is, aside from the performance effects, link-level data compression does not affect the operation of the higher-level protocols and applications.

The effectiveness of this type of compression is heavily data-dependent. Some objects, such as Postscript files, compress manyfold. However, it is also the case that many objects being transferred over the network have already been compressed at the application level. For example, most of the images which litter Web pages are encoded in GIF, JPEG, or other compressed format. The level of redundancy within the packets containing these objects is thus quite low. Therefore, transfers of these objects will not benefit much from additional data compression at the link level.

Furthermore, these compression schemes do not remove the redundancy of multiple transfers of the same object, because it occurs on too large a timescale to be observed by existing schemes. For example, gzip [5] searches for redundancy in windows of up to 32 KB, while correlation exists at 1000 times that scale, as we will show in Chapter 3.

2.1.2 Header Compression

Another type of compression that is frequently employed to use a link more efficiently is packet header compression. As with the data compression technique described above, header compression removes redundancy at the network level, and thus shares the same transparency benefits. However, while data compression removes redundancy within a packet, header compression removes redundancy between packet headers. Schemes specialized for compressing TCP/IP headers [10, 4] exploit the fact that half of the per-packet header information is likely to stay constant for the life of a TCP connection. These schemes therefore keep track of the header from the last packet encountered for each connection. Then, only the fields that have changed since the last packet are transmitted in the packet header.

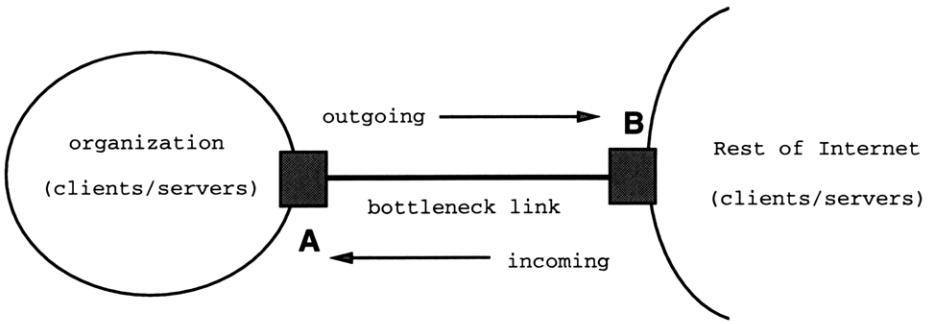

Figure 2-1: Generic Web Caching Configuration with a Bottleneck Link

These header compression techniques were motivated by the inefficiency of sending short packets across a link, as is often required by interactive sessions such as telnet connections. For these short packets, where the volume of each packet is occupied almost entirely by the header, compressing the headers may reduce their impact by an order of magnitude in the best cases, and hence may have a significant impact on bandwidth usage. However, for bulk data transfers, the volume of headers is small compared with the volume of payloads, i.e., even eliminating TCP/IP headers altogether would have only a small effect.

2.2 Application-Level Caching

An alternative to compressing at the link level is for each application to construct its own system for caching its objects. In a typical scheme, when a remote object is retrieved, a copy of the object is stored in a cache between the client and the server. Subsequent requests for that object are then served from the local cache rather than the remote source. This is clearly not viable for all applications, but may be worthwhile in terms of bandwidth for popular cases such as the Web.

To achieve the goal of more efficient use of network links, application-level caching taps yet another source of correlation in network traffic, that of repeated transfers of the same application object. Unlike link-level data compression, application-level caching does not depend on the data content of objects, rather it profits from redun-

dancy in traffic patterns.

However, while the goal of link-level compression is to improve the effective bandwidth of a particular link, application-level caching is aimed at improving wide area bandwidth utilization, as well as reducing client latency and distributing server load. Therefore, application-level caching presents a different set of tradeoffs than do the transparent link-level approaches. To examine the tradeoffs, we briefly consider Web caching using the generic configuration of Figure 2-1. Here, an organization is connected to the rest of the Internet by a single access link that is the bottleneck for transfers between the two domains.

Today's Web caches [17, 1] are deployed by organizations to reduce both client latency and wide area bandwidth requirements. A caching system may therefore be readily deployed at point A to protect incoming bandwidth by combining client requests. Existing caching systems, however, are more limited in their ability to protect outgoing bandwidth by combining server responses.

Also, it may not be possible to place a Web cache at point B and configure the rest of the Internet to use it, since point B is typically under the control of a different organization and proxy caches require the cooperation of their clients. That is, placement of application caches inside the network may require a large degree of sharing and cooperation between users compared to the link-level solutions we have studied, which may be deployed by the network operator when and as needed to buttress weak links.

Further, Web caches handle requests in a non-transparent manner. First, there is the cache consistency problem that a client may receive stale data from a Web cache. Also, since caching reduces the number of requests that actually reach the server, server operations such as request logging and access counting will be inaccurate. These issues may be compelling enough for clients to bypass the Web caches and contact the server directly or for servers to mark their objects as uncacheable.

2.3 Combining Compression and Caching

The system we briefly described in Section 1.2 combines some of the better aspects of compression and caching to make better use of the bandwidth of a heavily-loaded link. This system exploits the same source of redundancy that application-level caching does, i.e., repeated transfers of the same data, but removes the redundancy at the packet level, akin to link-level compression.

Since link layer schemes are transparent to applications, our system does not suffer from the undesirable side-effects present in application-level caching. For example, our scheme will never return stale data, nor complicate or bias server operations such as request logging.

Further, because they function across all applications, link layer solutions are capable of removing redundancy across multiple protocols, e.g., FTP as well as HTTP. More interestingly, our link layer solution suppresses identical content irrespective of application names and protocol details. For example, the same Web page contents will be suppressed, even if it is named by different URLs, generated dynamically, or marked as uncacheable. This effect may be significant: Douglis et al. found such duplication to occur for as many as 18% of full-body Web responses in some traces [6].

Our scheme complements application-level caching systems. Web traffic is so predominant that special-purpose caching mechanisms must become ubiquitous in order to distribute load and build a scalable Internet. Our scheme provides protection at a lower level and across changing application and traffic patterns. It can thus be applied to portions of the network selectively, e.g., to bottleneck access and long-haul backbone links, and will remove the replication that remains after application-level caching.

Chapter 3

Analysis of Replicated Traffic

The design of a system for suppressing replicated data transfers at the packet level depends on the characteristics of the replicated data itself. In this chapter, we present the results of a trace-driven analysis of replicated traffic. We begin with the goals of the analysis and the analysis procedure, and follow with the results themselves.

3.1 Goals of the Analysis

In 1993, Danzig et al. determined that a significant fraction of FTP traffic resulted from retransmissions of files [3]. Furthermore, many caching schemes such as [17, 1] have emerged following these findings to handle multiple requests for the same application objects, especially Web documents. The hit rates obtained by Web caches today [9] indicate that there is a significant number of redundant requests for Web documents.

However, even if repeated transfers of application objects were well characterized, it is not clear that they would translate into replication that can be detected and removed at the packet level. First, data sent multiple times must be parceled into packet payloads in the same manner, despite potentially different protocol headers, path maximum transmission units (MTUs), and protocol implementations. Second, the timescale of replication (which may be hours for Web documents) must be observable with a limited amount of storage.

Therefore, this chapter presents a new analysis aimed at characterizing replication at the packet level and the potential for designing a system which reduces the bandwidth consumption of the replication. In this analysis, we focus on replication of data payloads (i.e., packets minus the network headers). We consider a payload to be *replicated* if it is byte-for-byte identical to a previously encountered payload. We characterize the replication by answering the following questions:

- How much data is replicated? The amount of replication present in actual traffic gives a good indication of the bandwidth reduction one can expect from suppressing the replication.
- What type of packets are most likely to contain replicated data? This gives us insight into what sort of environment would benefit most from our scheme.
- How large are the packets containing replicated data? This provides an estimate of the savings that can be expected per packet by sending a small dictionary token in place of the data payload.
- What is the temporal distribution of replicated data? This indicates how much replication can be observed with limited amounts of storage.

3.2 Obtaining and Analyzing the Packet Traces

As input to our analysis, we collected a series of full packet traces of all traffic exchanged between our site and the rest of the Internet. New traces (rather than publicly available archives) were necessary because we require the entire packet contents in order to detect repeated data. Our site makes an interesting test case because it is a diverse environment hosting many client and servers, including the Web Consortium, MIT Laboratory for Computer Science and the MIT AI Laboratory.

Each trace was captured using `tcpdump` as a passive monitor listening to Ethernet traffic traveling on the segment between the Lab and the Internet. Five sets of 1-2 million packets each were gathered at different times of day, corresponding to approximately 2.6 GB of raw data in total. No packet capture loss was detected.

	All Inbound traffic			Inbound HTTP traffic		
Set	Total (MB)	Replicated (MB)	%	Total (MB)	Replicated (MB)	%
A	277	32	12	26	5	19
B	189	4	2	13	1	8
C	105	2	2	3	0.3	10
D	237	25	11	22	2	7
E	217	9	4	28	2	8
Total	1025	73	7	91	10	11

Table 3.1: Amount of replicated data in inbound traffic

	All Outbound traffic			Outbound HTTP traffic		
Set	Total (MB)	Replicated (MB)	%	Total (MB)	Replicated (MB)	%
A	554	98	18	267	63	24
B	563	117	21	384	109	28
C	294	61	21	239	57	24
D	606	116	19	420	103	25
E	594	134	23	427	123	29
Total	2610	526	20	1736	455	26

Table 3.2: Amount of replicated data in outbound traffic

We statically analyzed each trace by searching through the packets sequentially for replicated data. To expose the application data, we progressively stripped protocol headers up to the TCP/UDP level. For example, TCP payloads were identified by removing first the Ethernet, then IP and finally TCP headers. Our analysis therefore slightly underestimates the amount of replicated data due to changing headers at higher protocol layers that could not easily be taken into account; one important example of traffic that falls into this category is DNS responses.

3.3 Replication by Traffic Type

Our initial analyses classified replication by traffic direction (incoming and outgoing) and type (TCP, UDP, other IP, and other Ethernet). It quickly became evident that most replication occurred in outgoing TCP data on ports 80 and 8001, i.e., Web traffic

responding to queries from other sites. To highlight this, we separately classified TCP port 80 and 8001 traffic as HTTP traffic.

Tables 3.1 and 3.2 summarize the amount of replicated data that was found in each packet trace, for inbound and outbound traffic, respectively. The left-hand columns show the results for all types of traffic in each trace, while the right-hand columns summarize the replication in only the HTTP traffic for each trace.

These results support our intuition that there are significant amounts of replicated data that can be detected in network traffic. Furthermore, this type of analysis is useful in determining the links, as well as the direction over the link, for which suppression of replicated data can be effective.

For our traces, 20% (522 MB) of the outbound traffic, but only 7% (72 MB) of the inbound traffic, was identified as replicated traffic. Thus, for our link, suppression of replicated data would provide significant benefits, but only in the outbound direction. Therefore, in the remainder of this thesis, we focus on the outbound traffic over the link.

3.4 Replication by Packet Size

A further criterion that is important to our scheme is packet size. When fixed-length packet overheads and packet processing costs are taken into account, replication in large packets will result in a more effective system than replication in small packets.

To assess this effect, we classified the replicated data according to the length of the data payload being duplicated. Figure 3-1 depicts the cumulative volume of replicated data according to packet length. The sharp increases around 500 and 1500 bytes correspond to the default TCP segment size of 536 bytes and the maximum Ethernet payload size of 1.5 Kb. It is apparent that 97% of the volume of replicated data occurs in packets with a length greater than 500 bytes. This suggests that small per packet space costs required for compression will not result in a significant system overhead.

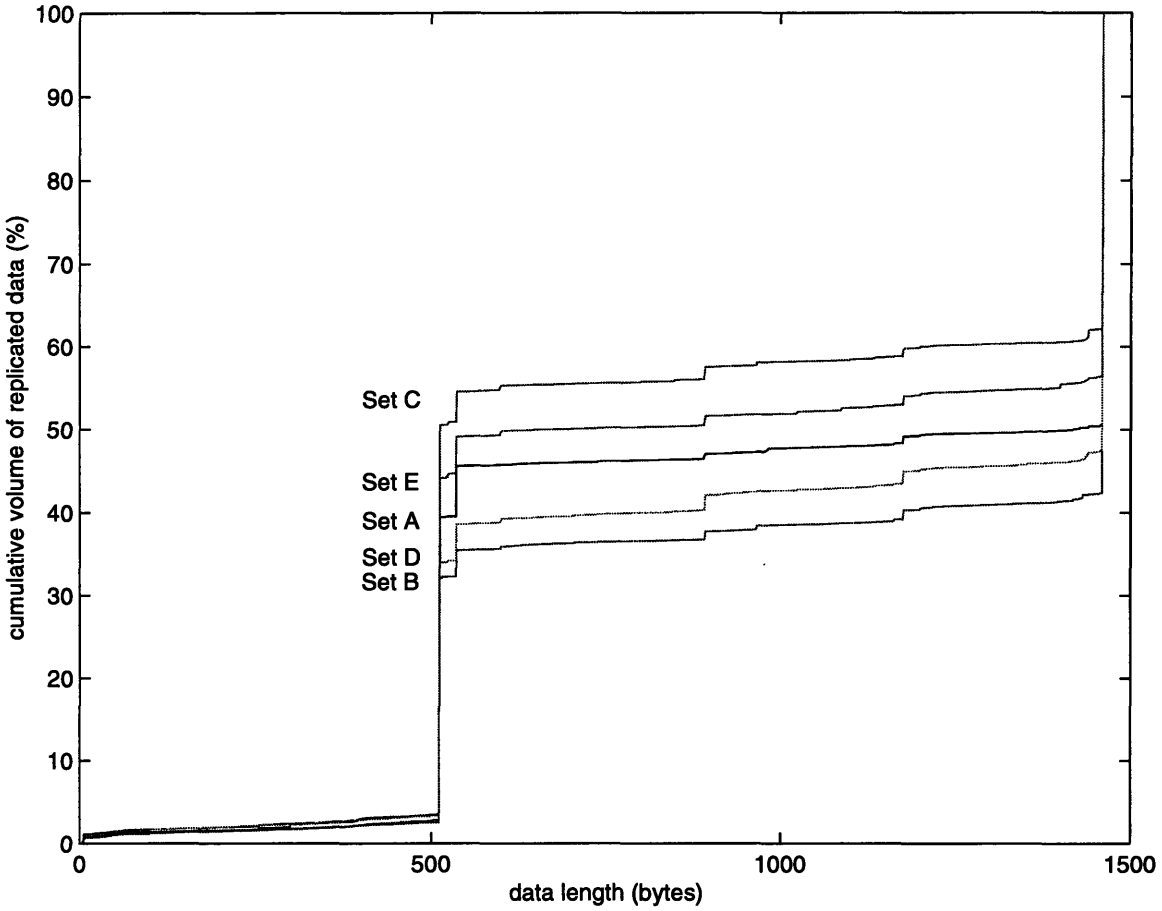

Figure 3-1: Cumulative volume of replicated data by packet length

3.5 Distribution of Replication

Finally, the timescale of replication events determines the size of the packet cache needed to observe and remove such redundancy. To quantify this effect, we determined the interval, in bytes of data, from each match to the previous copy of the match. These intervals were then grouped to compute the percentage of the replicated traffic that could be identified as a function of window size.

Figure 3-2 shows this result for all outbound traffic. The positive result that we infer is that the majority of replicated data can be observed with a cache of 200 MB, i.e., reasonable results can be expected if we cache the data in the amount of RAM that is presently available in PC's.

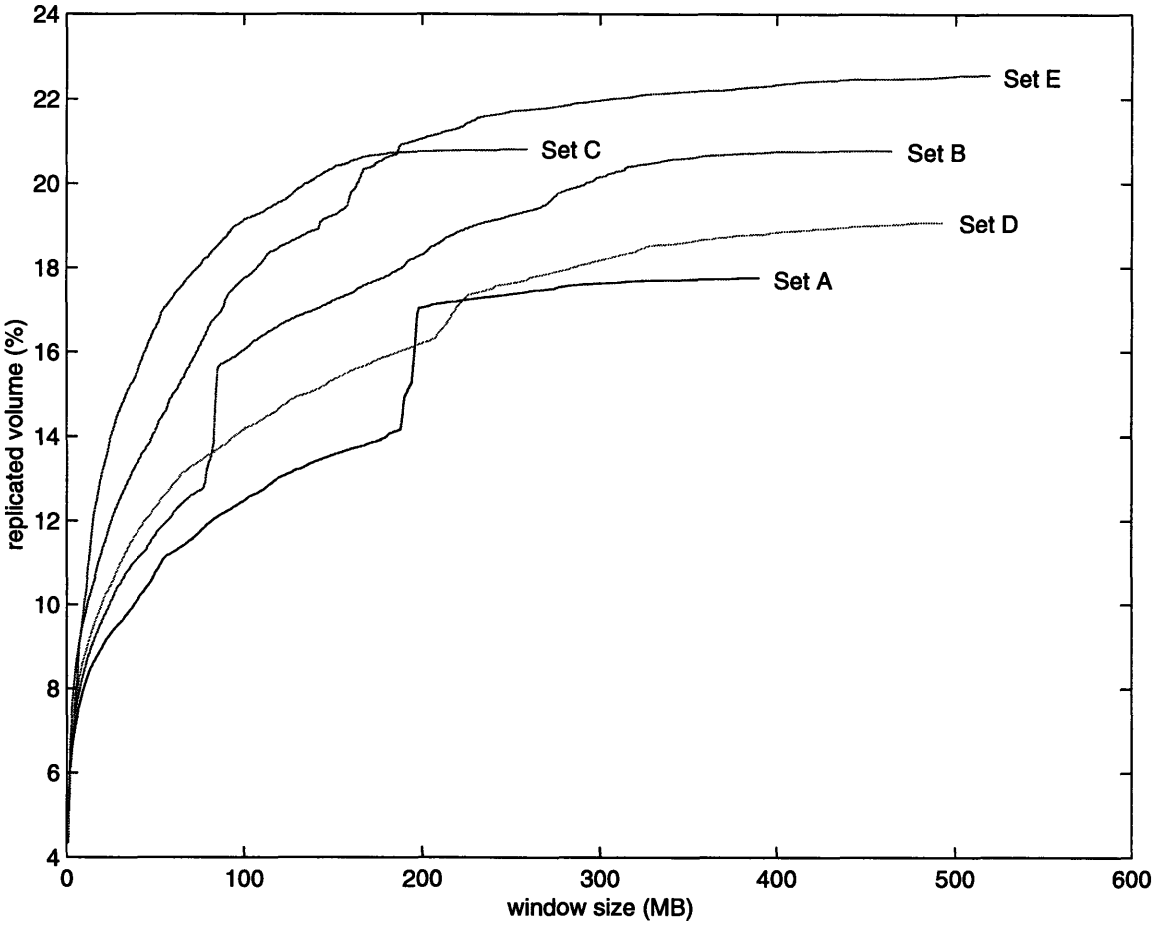

Figure 3-2: Percent of outbound traffic versus window size

3.6 Summary of Analysis

Through a trace-driven analysis, we characterized the packet-level replication of the network traffic entering and leaving our site. The results of the analysis indicate that there is significant replication at the packet level arising from replication and overlap at higher protocol layers. Furthermore, most of this replication was observed to be outgoing Web traffic, despite existing caching mechanisms. We also found that the bulk of the volume of replicated data resides in large packets, indicating that the savings expected from suppressing the replicated data considerably outweighs a small per-packet overhead. Finally, we have determined that a significant volume of replicated data can be observed in the 1-200 MB range, i.e., at a much larger

scope than existing compression schemes handle, yet at a small enough scope that the amount of memory in a PC is sufficient to identify the replication.

Chapter 4

System Design

In Chapter 3, we showed that a significant amount of replication exists in our network traces, particularly in the outbound traffic. We now describe, in detail, the design of a system architecture for suppressing the transfer of replicated data in order to make better use of the available link bandwidth.

4.1 Design Goals

The essence of our scheme is to transmit replicated data across the link as short dictionary tokens by using caches of data payloads at both ends of the link to maintain the dictionary and to encode and decode these tokens. In doing so, the system should:

- Add minimal space and time overhead to the packets being sent over the channel.
- Tolerate real-world conditions such as unreliable channels and machine failures.
- Provide a flexible architecture to allow the system to be tuned to particular environments or traffic patterns.

4.2 Dictionary Tokens

One issue which the system design must resolve is how to generate the dictionary tokens. A judicious choice of the algorithm used to generate the dictionary tokens is important with respect to minimizing the system overhead. In order to impart only a minimal overhead, the algorithm must:

- Produce small dictionary tokens. This reduces the per-packet overhead of sending dictionary tokens instead of the replicated data.
- Compute a dictionary token directly from the associated data payload. Since the data payload must be transferred over the link anyway, computing the token directly from the data means that no extra information needs to be conveyed across the link to create the same dictionary entry at each end of the link.

Our insight is that these requirements are well satisfied by cryptographic message digest algorithms, such as MD5 [14] or SHA [12]. These algorithms take as input an arbitrarily long sequence of bytes and produce as output a fingerprint of the input. We are aware of the use of fingerprints for identification and version control in various systems, e.g., DRP [16], but to the best of our knowledge this is the first time that fingerprints have been applied for this purpose at the network layer.

We selected the MD5 hash for our implementation because it is 128 bits long and may be calculated in one rapid traversal of the data; on a PentiumII (233MHz) the computational rate of fingerprinting exceeds 200 Mbps. Further, given that the hash is large enough and collisions (i.e., two distinct byte sequences hashing to the same value) rare enough, it is effectively a unique name for the data. For example, no collisions were detected in our trace data analysis. (Fingerprint collisions will be discussed in more detail in Section 4.5.2.)

4.3 Architecture

The main components of our architecture are shown in Figure 4-1, which shows a unidirectional compression system to simplify our description. The system consists of

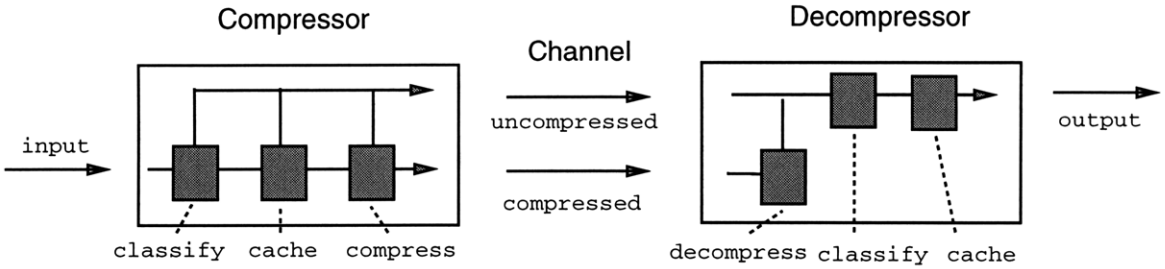

Figure 4-1: Components of the Architecture

a compressor, a channel, and a decompressor. The compressor is a repeater (perhaps part of a router) that accepts input traffic, processes it to compress replicated data, and transmits the resulting packets over the channel. Conversely, the decompressor accepts traffic from the channel, processes it to remove compression, and transmits it as output traffic. The channel itself may be any bidirectional link; we use the reverse direction to carry protocol control messages. Bidirectional compression is achieved by using two instances of the protocol, one for each direction.

Both the compressor and decompressor are composed of modules for classifying, caching, and compressing packets. Our architecture allows different policies to be selected for the implementation of each of these stages, subject to the constraint that compressor and decompressor implement identical processing in order to ensure that their dictionaries are closely synchronized. In particular, the dictionary caches must be of equal size. We describe each module in turn.

4.3.1 Classifying Packets

Not all packets need be entered into the dictionary cache. Our analysis in Chapter 3 showed that most of the replicated data in our traces was composed of outgoing Web traffic and large packets. An implementation may take advantage of such bias by selectively considering certain types of traffic for cache inclusion. The classification step in our architecture serves this role, and must be performed in the same manner at the compressor and decompressor.

The classifier further encodes the rules for identifying application data units (ADUs) embedded within the payload of packets, e.g., the stripping of headers up to the TCP/UDP level. By using application level framing concepts (ALF) [2], other extension policies could be designed to cater to specific application headers or compensate for the different division of data across different protocols.

4.3.2 Caching Policies

The cache module maintains the dictionary state, naming payloads by their fingerprint. Our architecture allows any fingerprint to be used depending on the required tradeoff between speed, space and collision resistance. In our implementation we use MD5, though stronger (i.e., more collision resistant) fingerprints such as SHA or weaker fingerprints such as MD4 may be used. Collisions will be discussed in Section 4.5.2.

Two policies govern the operation of the cache: the inclusion policy decides which payloads selected by classification should be admitted to the cache, and the replacement policy decides which payloads should be evicted when more space is needed. As for classification, the compressor and decompressor must implement identical policies.

Our default policies are simple: all payloads that are output by the classifier are entered into the cache, and the cache is maintained in least-recently-used order. For inclusion, an interesting policy would be to store replicated data only after its fingerprint had been encountered a certain number of times. Depending on the number of times a given payload is repeated, this may significantly reduce the storage required to suppress a given volume of replicated data. For replacement, results with Web caching [18] suggest that taking payload length into consideration may improve performance, since larger data payloads translate to higher per-packet savings.

4.3.3 Compression and Decompression

Finally, the compression and decompression modules exchange dictionary tokens to suppress the actual transfer of repeated data. Different policies may be used by the

compressor to decide when to compress payloads. Our default policy is to simply send tokens whenever repeated data is available. Alternative policies may be useful when the link possesses a large latency or high error rate and it is desirable to further reduce the chance that the far end of the link does not have the payload corresponding to a token. In these cases, it would be possible to send tokens after the payload has been sent multiple times, or, in the case of TCP traffic, send the token only after the acknowledgment of the payload is detected in the reverse direction.

4.4 Protocol Operation

We will now describe the exchange of messages between the compressor and decompressor under the ideal condition that no faults such as message loss, fingerprint collisions, or machine resets, occur. Modifications to handle these conditions will be discussed in Section 4.5.

The sequence of message exchange in the compression case is shown as a time sequence diagram (with time proceeding down the page) in Figure 4-2. These descriptions assume that the incoming packet passes the classification stage and satisfies the inclusion policy; packets that do not are simply forwarded over the link in the usual fashion.

When the compressor receives a packet {HdrA, X} to be forwarded over the link, where HdrA is the TCP/IP header and X is the data payload, it first computes H(X), the fingerprint of X. If it finds that no entry indexed by H(X) exists in its cache, it stores X in its cache, indexed by H(X). It then forwards the TCP/IP packet across the link. Upon receiving a TCP/IP packet forwarded over the channel, the decompressor also computes H(X), and stores X in its cache, indexed by H(X). The TCP/IP packet is then output from the system.

At some point later, the compressor may receive a packet {HdrB, X}, for which an entry indexed by H(X) already exists in its cache. This indicates that it has already received a packet containing X, which has been forwarded over the link. Therefore (assuming the compression policy is satisfied) it sends a packet to the decompressor

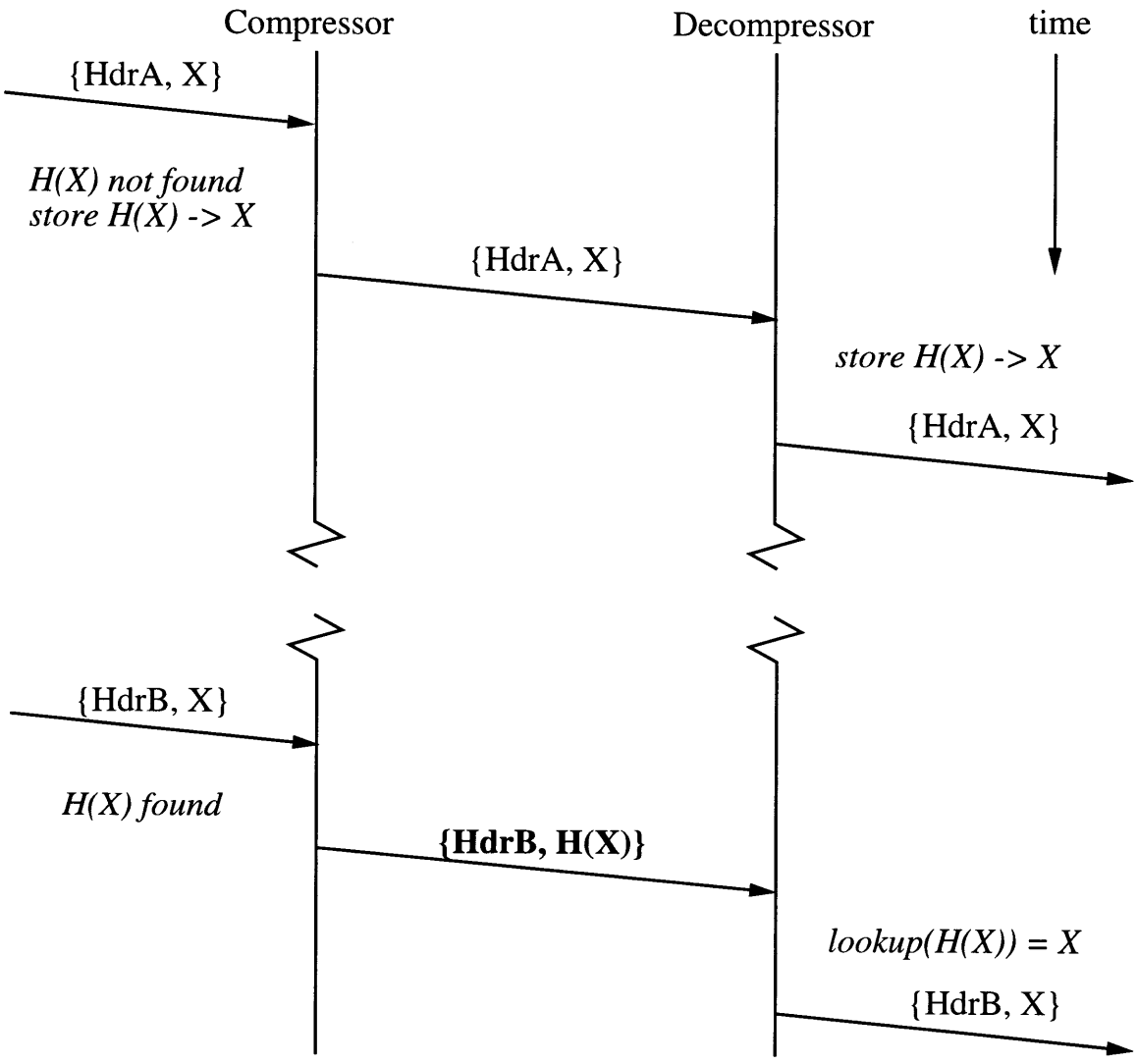

Figure 4-2: Compression protocol

containing the TCP/IP header HdrB and the fingerprint H(X). Fingerprint packets appear in bold type in the protocol diagrams.

The implementation must provide a means for these "fingerprint packets" to be distinguished from ordinary IP packets. In practice, this is not a problem, because the method used for demultiplexing protocols at the link level may be overloaded, e.g., in our implementation, we allocate additional types for the Ethernet protocol type field. Note that it is important that this identification scheme not increase the length of the packet, since this would necessitate a segmentation and reassembly protocol to accommodate maximum length datagrams.

When the decompressor receives a fingerprint packet {HdrB, H(X)}, it determines the data payload X that is indexed by H(X) in its cache. It then forwards the corresponding TCP/IP packet {HdrB, X} to the network on that end.

The net result of this protocol is that the same payload sent across the link multiple times will be transmitted as a short fingerprint packet after it has first been seen.

4.5 Robustness

Unfortunately, as with most distributed systems, there are certain real-world conditions which may have adverse effects on the operation of our system. We will now describe how the system can be made more robust with respect to some faults, and characterize the conditions in which the protocol fails.

4.5.1 Message Loss

Since our system will typically be implemented over an unreliable channel, there is a probability that a message will be lost by the channel. This could potentially cause the dictionaries to lose synchronization. To see this effect, consider the case where a packet containing the first instance of a data payload is lost while being sent across the link. This would cause a new dictionary entry to be created at the compressor, but not at the decompressor. The lack of feedback means that the compressor does not know that the decompressor never received the original payload. Therefore, it

will still send further copies of the payload by its fingerprint when the packet is retransmitted. However, the decompressor will not have an entry for the fingerprint in its dictionary.

As a result, if the protocol is left as is, we will never be able to send copies of the lost payload as long as the associated dictionary entry at the compressor exists. To correct this error, we introduce rejection handling into the protocol to handle events in which the decompressor receives a fingerprint that is not in its cache.

Figure 4-3 depicts rejection handling with another time sequence diagram. After message loss, if the decompressor receives a fingerprint packet {HdrB, H(X)} for which H(X) is not a valid entry in its cache, it sends the entire fingerprint packet (including the header) back to the compressor as a rejection packet. When the compressor receives this rejection, it determines the data X that is indexed by H(X). This is highly likely to be in the cache at the compressor since it was sent in the recent past. The compressor then sends the complete TCP/IP packet {HdrB, X} to the decompressor, which processes the packet as if it were receiving a new TCP/IP packet. It therefore enters it into its cache for subsequent use.

If any of the packets that are sent as part of the rejection handling are lost, or in the unlikely event that the compressor no longer has the payload corresponding to the rejected fingerprint in its cache, then the transmission has failed, and no further steps are taken to recover. This residual loss will then be handled by the reliability mechanisms of the application in the same manner that packet loss is handled today.

4.5.2 Fingerprint Collisions

We assume that the algorithm used to generate the fingerprints guarantees that the probability that two distinct inputs collide (i.e., yield identical fingerprints) is quite small. However, let us explore the effect of a collision in our scheme. Suppose that for an arbitrary fingerprint h, we have two data payloads, X and Y, such that $X \neq Y$ and $h = H(X) = H(Y)$, i.e., X and Y collide. Let us further suppose that a packet containing X has already passed through the system, adding a mapping from h to X in both dictionaries. Then, when a packet containing Y arrives at the compressor,

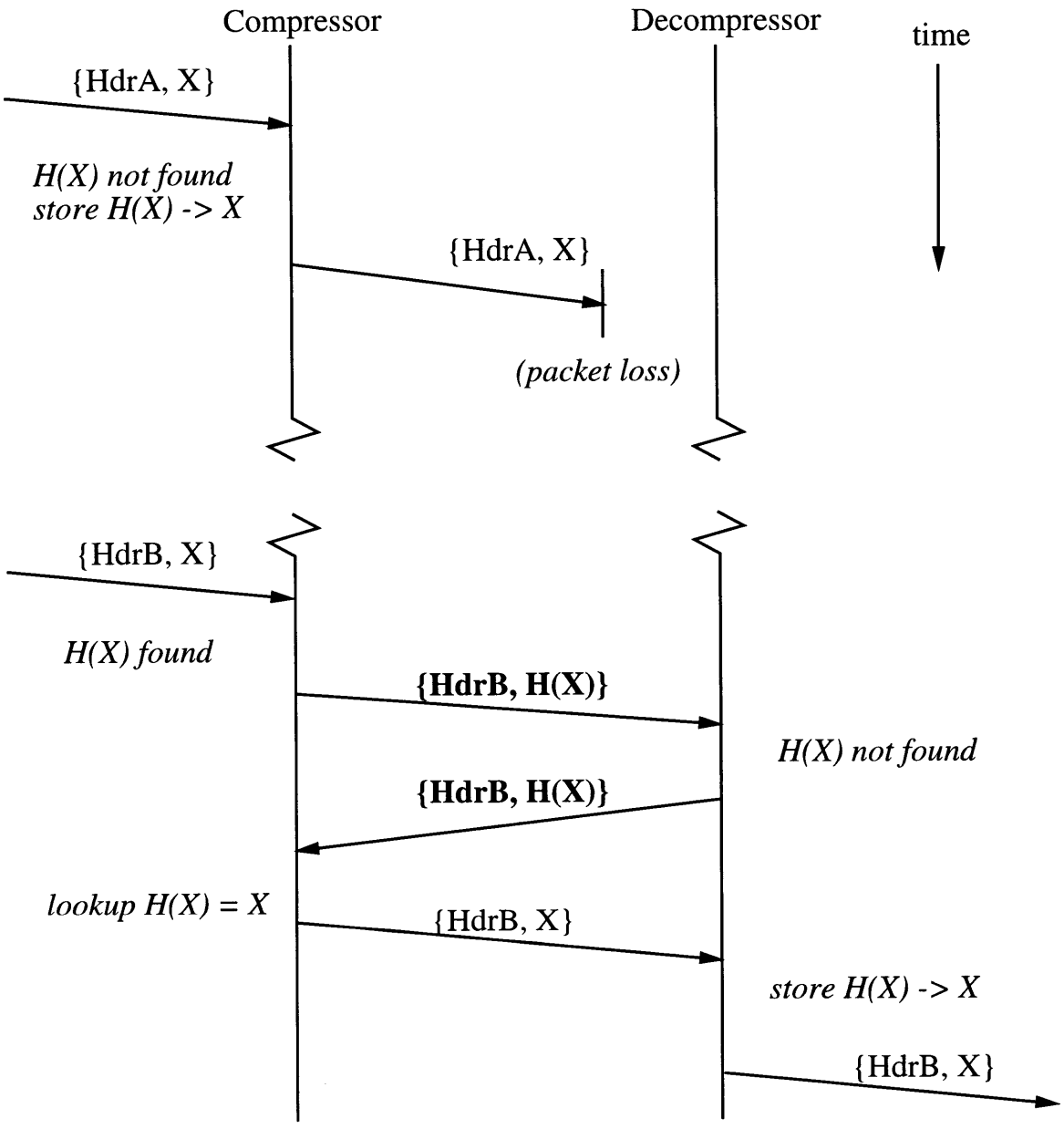

Figure 4-3: Rejection handling

its fingerprint h will be found in the dictionary, and sent across the link in lieu of Y. However, the associated value for h in the dictionaries is X, so the decompressor will expand h into X, rather than Y. Further, this error will occur for any packet containing Y that arrives at the compressor, as long as the entry for h exists in the dictionaries. Clearly, this is an undesirable effect, so although the probability of a collision is very small, we would like to provide a means to detect these collisions and avoid this scenario.

In our architecture, collision detection is performed as part of cache lookup and insertion at the compressor. Every time a fingerprint matches, the full payload data is compared with the existing cache contents before it is entered. If a collision is encountered, the fingerprint is marked as illegal in the dictionary and the colliding payload is transmitted without any compression. Any subsequent payloads which index to the illegal fingerprint are also transmitted uncompressed. These illegal entries must persist at the compressor until the decompressor is reset.

This collision detection mechanism makes the protocol robust with respect to fingerprint collisions, but only in the absence of message loss. To understand how fingerprint collisions can cause the protocol to fail in the presence of message loss, we consider the following scenario. Suppose that a mapping from h to X exists in the decompressor's cache, but not the compressor's cache. This state can be reached if an entry is flushed from the compressor's cache, but the loss of a message has allowed the corresponding entry to remain in the decompressor's cache. Now, suppose a colliding payload Y, where $H(Y) = h$, arrives at the compressor, but is lost by the channel. Then, when another copy of Y arrives at the compressor, it is sent by its fingerprint h across the channel. The decompressor thus incorrectly expands h into X.

The probability that this condition will occur is expected to be quite low. More specifically, the probability is upper-bounded by the probability that a fingerprint collision occurs and the colliding payload is lost by the channel.

4.5.3 Bit Errors

When packets are being sent over a link, certain environmental conditions may cause bits to change. In many cases, this translates into packet loss because a link-level checksum subsequently fails. However, it is possible that the checksum does not detect the bit errors, and the corrupted packet is processed as normal. We now investigate the effects of these corrupted packets.

The corrupted data will yield a different fingerprint than the original data, so a different, but valid, mapping will be created at the decompressor. As a result, the decompressor may have an entry that the compressor does not. This state is similar to one that can occur when an entry is flushed at the compressor, but a message loss allows the corresponding entry to remain at the decompressor. Thus, in the absence of fingerprint collisions, this does not affect the correctness of the protocol. However, in the presence of fingerprint collisions, we are faced with the same problem described in Section 4.5.2.

Furthermore, it is conceivable that bit errors can cause the payload to mutate a colliding payload (i.e., one whose fingerprint collides with a different payload in the cache). This can cause the decompressor to have an entry that conflicts with one in the compressor. Thus, whenever the compressor resends the original payload as a fingerprint, it will be incorrectly expanded by the decompressor.

The probability that either of these failure modes occurs is at most the probability that a bit-error causes a fingerprint collision. This is expected to be extremely low.

4.5.4 Machine Resets

Inevitably, either the compressor or decompressor will at some time be reset. In this case, it is desirable to also reset the state of the other machine. This is accomplished by sending a reset message to the other machine.

In the event that the compressor is reset during the operation of the protocol, it is necessary to also reset the state of the decompressor. For instance, the record of illegal fingerprints is lost when the compressor is reset. Therefore, we must ensure

that the decompressor does not have any entries indexed by a previously illegal fingerprint. Thus, when the compressor is reset, it repeatedly sends a reset message to the decompressor until the decompressor acknowledges that it has in fact been reset.

Further, resetting the decompressor during operation of the protocol may result in significant rejection traffic, as the compressor may send many fingerprints for which the decompressor no longer has a dictionary entry. This rejection traffic can be avoided if the dictionary at the compressor is reset as well. Therefore, whenever the decompressor is reset, it explicitly sends a reset message to the compressor. Note that this is merely a performance optimization, and is not essential for correctness.

4.5.5 Memory Corruption

Memory corruption can cause the caches to contain invalid mappings of fingerprints to payloads. As a result, the decompressor may always incorrectly expand a fingerprint. We expect memory corruption to be extremely rare, and do not provide any means of tolerating it.

Chapter 5

Experimental Results

To evaluate the system, we developed an implementation and performed three sets of experiments:

- We measured the bandwidth savings that our system provides in practice when operating on real traffic.
- We measured the baseline performance of the compressor and decompressor to gauge at what rates our system may be used.
- We compared the bandwidth savings produced by our system with alternative compression schemes.

5.1 Implementation

We implemented the architecture described above using a pair of Intel-based PentiumII 300MHz machines running Linux 2.0.31. The machines were directly connected to each other via a dedicated 10 Mbps Ethernet. Each machine was also connected to the 100 Mbps Ethernet network which comprises our research group's LAN. Both compressor and decompressor modules were written in C and ran as user-level processes.

The compressor machine was configured with IP forwarding enabled in the kernel. However, we modified the kernel forwarding routine to send the packets to the user-

level program instead of handling the routing itself. We also allocated additional Ethernet protocol types to distinguish the fingerprint and rejection packets from the uncompressed IP packets.

We implemented the dictionary caches using hash table structures with a least-recently-used replacement strategy. For fingerprints, we used the MD5 hash of the payload. We also used a classifier that only accepted data with payloads of at least 500 bytes since our earlier analysis indicates that the remaining data comprises only 3% of the replicated volume. Finally, we limited the amount of memory available for the caches to 200 MB each.

5.2 Bandwidth Reduction

The main design goal of the system is to reduce the amount of bandwidth consumed by replicated data. We quantified the effectiveness of our system by sending the outbound traffic of the traces from Chapter 3 through the system and measuring the resulting bandwidth savings. On average, the system saved about 16% of the bandwidth, which is 80% of the replicated traffic detected in Section 3.3.

We then inserted our system into the network at the point where we previously gathered traces; see section 3.2. We kept track of the amount of data input to and output from the system and the amount of data transmitted across the compressed channel while the system ran for 24 hours and processed approximately 50 GB.

Figure 5-1 shows the resulting bandwidth reduction for each minute of the run. It shows that the implementation is effective in reducing the bandwidth utilization by approximately 20% on average. Note that this is higher than the savings observed for the packet traces. We believe that this is because the traces are too short to capture some of the replication, and thus do not fully represent the amount of replication present in the network traffic.

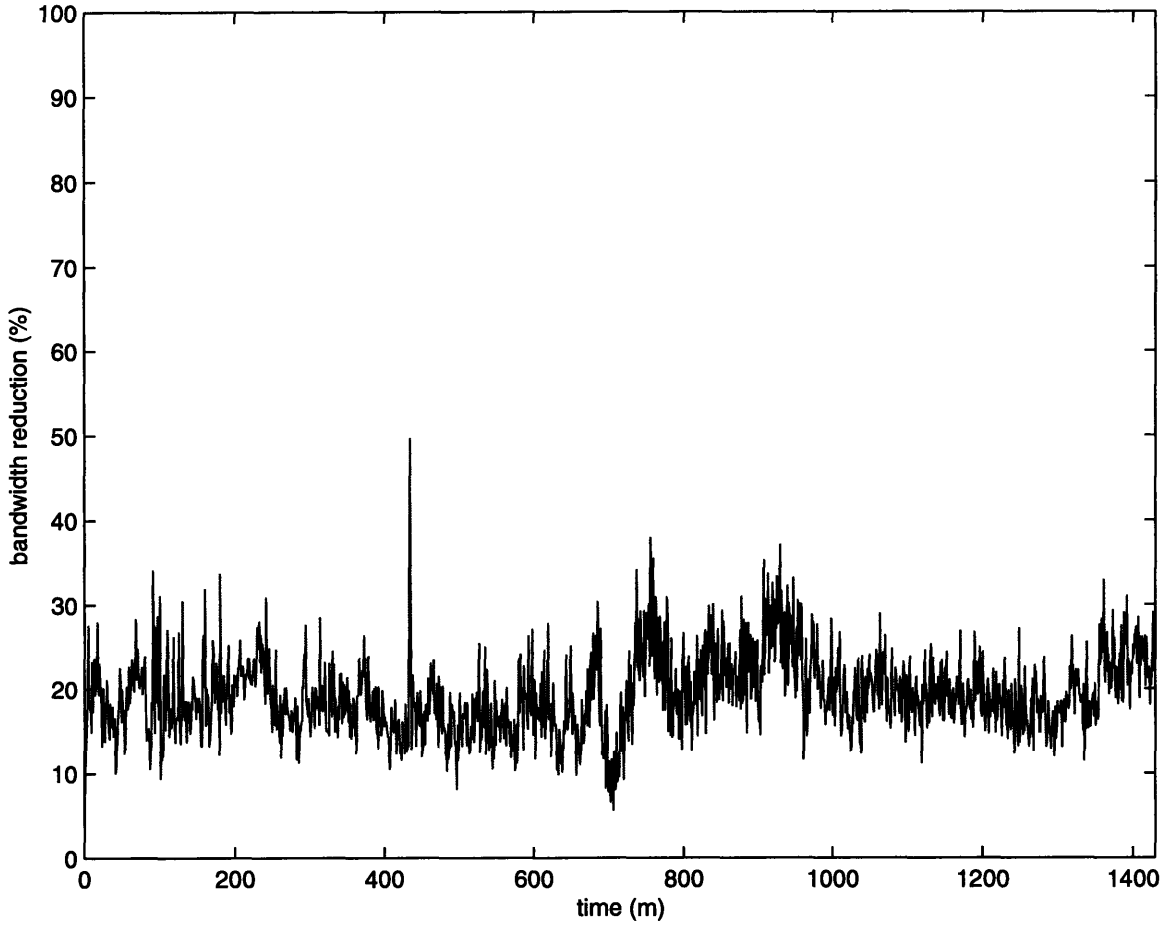

Figure 5-1: Bandwidth reduction of all outbound traffic

5.3 System Performance

Since we are interested in the potential of this scheme for use in higher speed networks (with capacities exceeding 10 Mbps) we measured the overall system performance to see how fast it would run.

We measured throughput by running the system over a 100 Mbps channel and using a traffic source that generated a stream of packets containing no repeated payloads. Note that copying new payloads into the cache requires the same number of operations as does collision detection. Therefore, the packet stream described above represents the worst load for our system. We drove the system with a TCP connection between source and sink (that flowed over the channel), making use of

Set	Suppression (%)	Compression (%)	Both (%)
A	12.08	20.09	31.30
B	15.50	24.08	37.35
C	18.81	18.42	33.90
D	14.37	17.95	32.44
E	17.90	18.58	35.32
Avg	15.73	19.92	34.07

Table 5.1: Bandwidth effects of Suppression and Compression (gzip) on outbound traffic

TCP's congestion control mechanisms, to probe the maximum practical throughput.

We measured latency by using `tcpdump` as a passive monitor to capture and timestamp packets entering and leaving both the compressor and decompressor. To observe small latencies, we used a modified Linux kernel that recorded timestamps using the processor cycle counter at driver interrupt time.

The results of our tests were that our implementation was capable of forwarding over 6000 packets per second with a maximum throughput exceeding 60 Mbps. Furthermore, the latencies of the compressor and decompressor were both approximately 390μs.

These results are encouraging; our system can already run at rates exceeding T3 (45 Mbps), despite the fact that it is a user-level prototype that has not been tuned, e.g., to minimize copies. Further, preliminary comparisons with other compression schemes (such as gzip as discussed below) suggest that our scheme is significantly less computationally expensive. The similar and low latencies of compressor and decompressor result in a balanced system for given hardware and a small impact on overall latency. They are also likely to improve significantly with a kernel-space implementation since the overhead of context switching would be removed.

5.4 Other Compression Methods

Since bandwidth savings are heavily data-dependent, we compared our bandwidth reductions with those of other compression schemes to place them in context and

help gauge their significance.

As an approximation to real systems, we ran our trace data through a process that applied gzip [5] compression to packet payloads and recorded volume statistics. To simulate useful schemes under real-time and high throughput conditions, we used the fastest library setting and processed packets individually; even so, gzip is substantially slower than our scheme and could not keep pace with a 10 Mbps link. Table 5.1 compares this compression with our scheme for removing replicated data. We infer from these results that our scheme provides similar benefits, somewhat smaller on average, but requiring far less computation.

To look at the effects of combining our suppression with regular compression, we ran our trace data through a process that combined the two, first removing replicated data and compressing the remainder. Table 5.1 also shows these results. It highlights the fact that suppression and compression are complementary, as each tends to tap correlation on different timescales.

We also considered the impact of header compression, but quickly realized that it would provide smaller savings. With the average packet size of our trace close to 500 bytes, elimination of TCP/IP headers from all packets would save no more than 8% of the bandwidth, and this best case is unlikely to be obtained across a link where there is significant traffic mixing.

5.5 Summary of Results

Through experimentation on a prototype implementation, we evaluated the practicality of our system. We determined that the system is effective at reducing bandwidth utilization by about 20% in practice, which is more than our preliminary analysis led us to believe. Furthermore, we demonstrated that the system is capable of running at speeds exceeding T3 (45 Mbps), which is much faster than other compression techniques. Finally, we showed that although our system provides similar benefits to other compression techniques, even greater reduction rates can be achieved by combining our suppression scheme with compression techniques.

Chapter 6

Conclusions and Future Work

In this thesis, we presented an innovative link compression protocol that suppresses the transfer of replicated payloads. The protocol itself works by maintaining (nearly) synchronized packet caches at either end of a link and sending fingerprints in place of repeated payloads. We have demonstrated that, despite existing caching mechanisms, there is a significant amount of replicated traffic that is amenable to detection and reduction by our scheme. We have further shown by experimentation that the protocol is lightweight enough to be implemented on commodity hardware at rates exceeding T3 (45 Mbps), and that for real packet traces, the increase in available bandwidth from our scheme can be around 20%. This makes it an economically viable option for increasing available Internet access bandwidth.

6.1 Contributions and Conclusions

This thesis has made three major contributions involving replication in network traffic at the packet level:

- **Analysis of Replicated Data:** Through the use of packet trace analysis, we have shown that there is significant replication of data at the packet level. Furthermore, the traffic was characterized in terms of the amount of replication present, the type of data that is likely to be replicated, and the temporal dis-

tribution of the data. To our knowledge, there have been no previous studies of replication at the packet level.

- **Design of a Packet-Level Caching and Compression Scheme:** A novel link layer caching and compression system was designed and implemented. This system exploits packet-level replication to reduce bandwidth consumption.

- **Economical System Implementation:** The system was implemented on inexpensive memory and general-purpose processors. This implementation has demonstrated that significant bandwidth savings can be achieved in practice at useful rates.

In today's network traffic, significant redundancy exists at several levels and can be exploited to build better networks in both theory and practice. Existing compression schemes have shown how to use the redundancy at the byte level, while existing caching techniques deal with the redundancy at the application level. Our protocol is novel because it exploits redundancy at the packet level. In doing so, our protocol exhibits several interesting properties:

- Unlike other compression methods, it is independent of the format of packet data contents, and so provides benefits even when application objects have been previously compressed.

- It utilizes a source of correlation that is neither available at individual clients and servers nor found by existing link compression schemes, and hence can be used in addition to other link compression schemes.

- It provides the bandwidth reduction benefits of caching in a transparent manner, e.g., unlike Web caching there is no risk of stale information or loss of endpoint control.

- Unlike Web caching, it does not depend on particular protocols, client configuration or application names; it may thus be useful as a general-purpose mechanism to protect links from redundant transfers (which have many sources) as applications and traffic patterns change.

Another interesting observation arises from our innovative use of fingerprints. We have demonstrated that fingerprints can provide a convenient and efficient handle for manipulating objects within a network. Furthermore, these handles are independent of upper-level details, thus they can be used across many different applications.

6.2 Future Work

The results presented in this thesis suggests that our approach of suppressing replicated data at the packet level can lead to significant reductions in bandwidth utilization. There are several areas that merit further work:

- **Analysis of Replication at Other Sites:** In our analysis, we used the traffic between the MIT Laboratory for Computer Science and the rest of the Internet to reach our conclusions. It would be valuable to perform similar analysis of replication at various other sites to more fully understand the potential of our system.

- **Classification Techniques:** In our implementation, the classifier simply identified TCP packets with payloads of at least 500 bytes. The development of additional classification techniques that increase the amount of data that we are able to detect as replicated may improve the effectiveness of the system as a whole.

- **Caching Policies:** Other system parameters for which further investigation could improve the value of the system are the cache insertion and replacement policies. Our current implementation includes everything the classifier outputs in the cache and uses a least-recently-used replacement policy. There is reason to believe that more interesting policies would improve the range of match detection for a given amount of storage.

- **Effects of Error Rates:** Note that with our system architecture, the loss of a packet may cause additional messages to be transferred to handle a subsequent

packet, thus increasing the probability that the subsequent packet will be lost as well. Since the bit error rates of our channel were so small, this effect was negligible. However, the impact of the protocol on performance should be characterized across a range of bit error rates to confirm that it does not exacerbate packet loss.

- **Robustness of Design:** We have shown in Section 4.5 how we have modified our basic scheme to be more robust with respect to certain faults. We also have described how some conditions will cause the protocol to fail. It would be useful to investigate solutions to these problems, and to characterize their effects on the performance of the system.

6.3 Perspective

Today, the Internet has largely become an information retrieval medium, and is likely to remain so. As such, the amount of redundant information being transferred over the network has become so significant that we must learn how to take advantage of it to build better networks. This thesis takes a new step in that direction by introducing an innovative approach that improves network utilization by exploiting redundancy that is typically missed by existing compression and caching schemes.

Bibliography

[1] A. Chankuntod et al. A Hierarchical Internet Object Cache. In *USENIX'96*, 1996.

[2] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM '90*, 1990.

[3] P. Danzig et al. A Case for Caching File Objects Inside Internetworks. In *SIGCOMM '93*, 1993.

[4] M. Degermark et al. Low-loss TCP/IP Header Compression for Wireless Networks. In *MOBICOM'96*, 1996.

[5] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. Request For Comments: 1951, May 1996.

[6] F. Douglis et al. Rate of Change and other Metrics: a Live Study of the World Wide Web. In *USENIX Symp. on Internetworking Technologies and Systems*, 1997.

[7] D. Ewing et al. A measurement study of internet file transfer traffic. Technical Report CU-CS-571-92, Department of Computer Science, University of Colorado, Boulder, CO, January 1992.

[8] National Laboratory for Applied Network Research (NLANR). Tutorial: Insight into current internet traffic workloads, 1997. http://www.nlanr.net/NA/tutorial.html.

[9] National Laboratory for Applied Network Research (NLANR). Hierarchical caching system usage statistics, May 1998. http://ircache.nlanr.net/Cache/Statistics/Hierarchy.

[10] V. Jacobson. Compressing TCP/IP Headers for Low-Speed Serial Links. Request For Comments: 1144, February 1990.

[11] Network Wizards. http://www.nw.com.

[12] NIST. Secure Hash Standard. FIPS PUB 180-1, May 1993.

[13] D. Rand. The PPP Compression Control Protocol. Request For Comments: 1962, June 1996.

[14] R. Rivest. The MD5 Message-Digest Algorithm. Request For Comments: 1321, April 1992.

[15] W. Simpson (Ed.). The Point-to-Point Protocol. Request For Comments: 1661, August 1994.

[16] A. van Hoff et al. 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.

[17] D. Wessels. The Squid Internet Object Cache. http://squid.nlanr.net/Squid/, 1997.

[18] S. Williams et al. Removal Policies in Network Caches for World-Wide Web Documents. In *SIGCOMM '96*, 1996.

[19] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.