Increasing Effective Link Bandwidth by Suppressing Replicated Data

Jonathan Santos and David Wetherall
Massachusetts Institute of Technology

Increasing Effective Link Bandwidth by Suppressing Replicated Data [*]

Jonathan Santos [†]
David Wetherall [‡]
Software Devices and Systems Group
Laboratory for Computer Science
Massachusetts Institute of Technology
http://www.sds.lcs.mit.edu/

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 utilized inexpensive commodity hardware to design a novel link layer caching and compression scheme that reduces bandwidth consumption. Our scheme is motivated by the prevalence of repeated transfers of the same information, as may occur due to HTTP, FTP, and DNS traffic. Unlike existing link compression schemes, it is able to detect and use the long-range correlation of repeated transfers. It also complements application-level systems that reduce bandwidth usage, e.g., Web caches, by providing additional protection at a lower level, as well as an alternative in situations where application-level cache deployment is not practical or economic.

We make three contributions in this paper. 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 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).

[*]This work was supported by DARPA, monitored by the Office of Naval Research under contract No. N66001-96-C-8522.

[†]Email: jrsantos@mit.edu

[‡]Email: djw@lcs.mit.edu

1 Introduction

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. For example, access links (modem, ISDN, T1, T3) restrict bandwidth due to cost, while wireless links restrict bandwidth due to properties of the media. A traditional solution to this problem is the use of data compression, either at the link or application level. Existing compression schemes, however, tend to miss the redundancy of multiple instances of the same information being transferred between different clients and servers. This is problematic because such transfers have become prevalent with the growth of information services such as the Web.

Danzig's 1993 study of Internet traffic [3] noted that half of the FTP transfers could be eliminated with a caching architecture that suppressed multiple transfers of the same information across the same link. Since that time, 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 perceived to be high, despite the application-level caching mechanisms that have emerged to curtail it.

In this paper, we revisit the problem of improving effective link bandwidth in the context of traffic with replicated data, as may occur due to TCP retransmissions, application-level multicast, DNS queries, repeated Web and FTP transfers, and so on. We have designed an innovative link compression scheme that uses a network-based cache to detect and remove redundancy at the packet level. Our scheme takes advantage of the availability of inexpensive memory and general-purpose processors to provide an economical means of purchasing additional bandwidth. That is, given the one-time costs of \$5000 per PC and the monthly costs of \$2500 per T1 (1.5 Mbps), it is cheaper to purchase the two PCs used by the scheme than the bandwidth they are expected to save.

Our scheme has several interesting properties:

- It is independent of the format of packet data contents and so provides benefits even when application objects have been previously compressed, e.g., for Web images already in JPEG or GIF format.
- It utilizes a source of correlation 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, e.g., there is no risk of stale information or loss of endpoint control.
- It constructs names at the link level using fingerprints and so does not depend on higher level protocol names or details. For example, the same information identified by different URLs will be compressed by our scheme, but not by Web caches.

Our scheme overlaps application-level caching systems – most notably Web caches – in that both reduce the impact of repeated transfers of the same information. However, our scheme is intended to complement Web caches rather than to compete with them, since it addresses a slightly different goal and works at a different level. For example, Web caches do not take advantage of replication across multiple caching systems, protocols and application objects.

In this paper, we present: a trace-driven traffic analysis that motivates our scheme; the design of our system; and an experimental characterization of a prototype implementation. Our traffic analysis in Section 2 uses several traces of at least one million packets each that we recorded between our site (the MIT Laboratory for Computer Science, including the Web consortium) and the rest of the Internet. In Section 3, we describe the system architecture and compression protocol, along with a prototype implementation running under Linux. In Section 4, we evaluate the performance of this prototype. We then contrast our system with related work and conclude in Sections 5 and 6, respectively.

2 Analysis of Replicated Traffic

To understand the potential of a system for suppressing replicated data transfers at the packet level, we began our design by analyzing network traffic. We define a packet to be *replicated* when the contents of its payload match exactly the contents of a previously observed payload. Since packet headers are expected to be constantly changing and a function of the source and destination hosts rather than the data being transported, we do not consider them in our search for replicated data.

Note that it is not clear that overlapping Web transfers will translate into replication that satisfies our definition and that may 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. We therefore characterize the replication as defined above by answering the following questions:

- How much data is replicated?
- What kind of data is most likely to be replicated?
- What is the temporal distribution of replicated data?

2.1 Obtaining 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. The choice of our site was expedient, but it makes an interesting test case because it is a diverse environment hosting many

	All Inbound		Inbound HTTP		All Outbound		Outbound HTTP	
Set	Total Vol. (MB)	% Repl.	Total Vol. (MB)	% Repl.	Total Vol. (MB)	% Repl.	Total Vol. (MB)	% Repl.
A	277	12	26	19	554	18	267	24
B	189	2	13	8	563	21	384	28
C	105	2	3	10	294	21	239	24
D	237	11	22	7	606	19	420	25
E	217	4	28	8	594	23	427	29
Total	1025	7	91	11	2610	20	1736	26

Table 1: Total volume and replicated percentage (by volume) of inbound and outbound traffic

clients and servers. It includes 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.

2.2 Analysis Procedure

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 example of traffic that falls into this category is DNS responses.

2.3 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

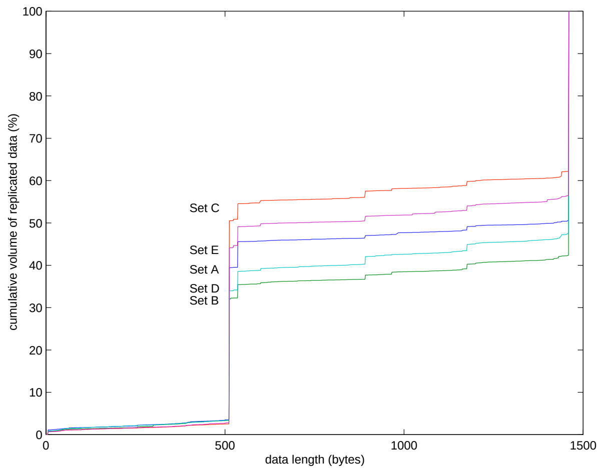

Figure 1: Cumulative volume of replicated data by packet length

and 8001 traffic as HTTP traffic.

Table 1 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 present in the traces. Further, most of the traffic, as well as a greater percentage of replication, exists in the outbound traffic. Therefore, for the remainder of this paper, we will focus on the outbound traffic over the link.

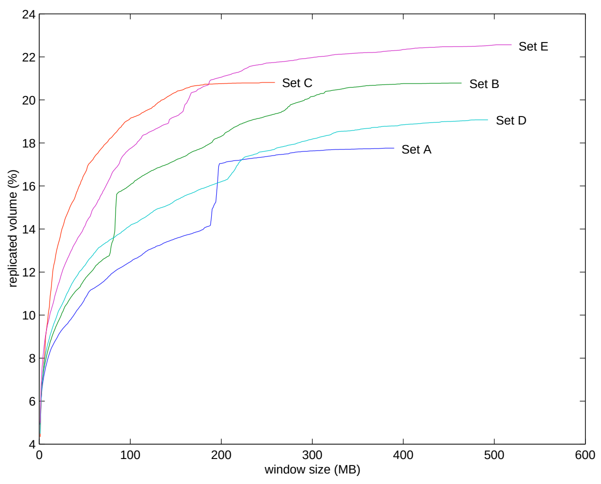

Figure 2: Percent of outbound traffic versus window size

2.4 Replication by Packet Size

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

To assess this effect, we classified the replicated data according to the length of the data payload. Figure 1 depicts the cumulative volume of replicated data according to packet length. The sharp increases round 500 and 1500 bytes correspond to the default TCP segment size is 536 bytes and the maximum Ethernet payload 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.

2.5 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 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 PCs.

3 Design and Implementation

We now describe the design and implementation of a compression architecture that suppresses replicated data based on the analysis from Section 2. The overall goal of our scheme is simply to transmit repeated data as a short dictionary token, using caches of recently seen data at both ends of the link to maintain the dictionary and encode and decode these tokens.

The correct operation of this scheme as a distributed system is complicated by the fact that messages may be lost by the channel. Our design must resolve the following issues:

- How are dictionary tokens generated?
- How are dictionaries at either end of the link maintained in a (nearly) synchronized state?
- How are (inevitable) differences in dictionary state handled?

Our approach is based on the insight that the fingerprint of a data segment is an inexpensive name for the data itself, both in terms of space and time. We are aware of the use of fingerprints for identification and version control in various systems, e.g., Java RMI/OS, 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 [12] for our implementation because it is 128 bits 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 rare enough, it is effectively a unique name for the data. For example, though our architecture handles collisions, none were detected in our trace data analysis.

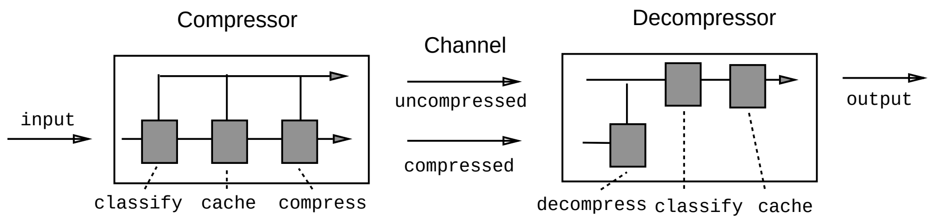

Figure 3: Components of the Architecture

To handle message loss in a lightweight fashion, we have opted to process messages independently, such that each message is the unit of error generation and recovery. That is, our scheme is connectionless (aside from the dictionary state) and does not require that a reliable transport protocol be run across the link in order to recover from errors.

3.1 Architecture

The main components of our architecture are shown in Figure 3, which shows a unidirectional compression system to simplify our description. The system consists of 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 several 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.

3.1.1 Classifying Packets

Not all packets need be entered into the dictionary cache. Our analysis in section 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 for specific application headers or compensate for the different division of data across different protocols.

3.1.2 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 collisions. In our implementation we use MD5, though stronger fingerprints such as the SHA [10] or weaker fingerprints such as MD4 may be used.

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 [15] suggest that taking payload length into consideration may improve performance, since larger data payloads translate to higher per-packet savings.

A further issue that affects inclusion is fingerprint collision. Collisions are expected to be extremely rare, but nevertheless it is conceivable that they may occur. If so, they must not result in a deterministic error, with the same offending data being repeatedly transferred to correct perceived transmission errors.

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.

3.1.3 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 when the acknowledgment of the payload is detected in the reverse direction.

3.2 Protocol Operation

We now describe the exchange of protocol messages between the compressor and decompressor. These fall into three cases.

- In the normal case, a payload is transferred (being entered in the dictionary as a side-effect) and after some interval another payload with the same contents is transferred, this time as a dictionary token. We refer to this case as *compression*.
- Occasionally, however, message loss on the channel may cause the two caches to lose synchronization and a dictionary token that is transferred must be returned to the sender to be resolved. We refer to this case as *rejection*.
- Further, if either the compressor or decompressor is restarted during the operation of the protocol, it is desirable to reset the other cache to a known state. Therefore, we add reset messages to the protocol.

3.2.1 Compression

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. 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

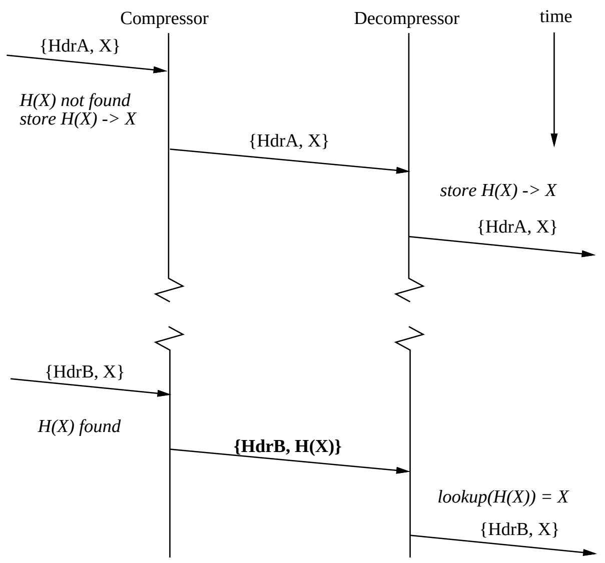

Figure 4: Compression protocol

already received a packet containing X, which it forwarded over the link. Therefore (assuming the compression policy is satisfied) it sends a packet to the decompressor containing the TCP/IP header HdrB and the fingerprint H(X). Fingerprint packets appear in bold type in the protocol diagrams.

The implementation must therefore provide a means for these "fingerprint packets" to be distinguished from ordinary IP packets. In practice, this is not a problem, because the codepoint used for demultiplexing protocols at the link level may be overloaded, e.g., 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.

3.2.2 Rejection

The protocol as described above is incomplete, for it does not handle the case where a packet contain-

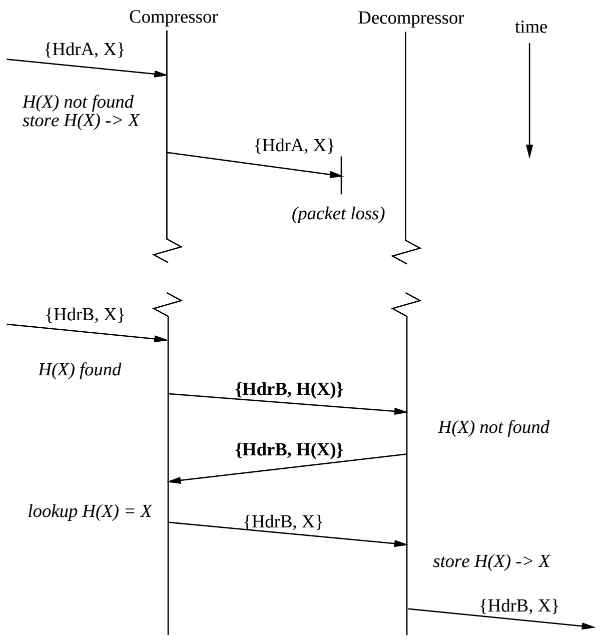

Figure 5: Rejection handling

ing the first instance of a data payload is lost while being sent across the link. We expect this case to be rare for most channels, since bit error rates typically contribute negligibly to the overall packet loss, and loss due to congestion may be detected at the compressor (since it results in queue overflow) and the lost payloads not entered into the dictionary.

Nevertheless, if the protocol is left as is, the lack of feedback means that the compressor does not know that the decompressor never received the original payload. This means that it will send further copies of the payload by its fingerprint when the packet is retransmitted, causing ongoing loss. 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 5 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.

3.3 Reset Messages

During normal operation of the protocol, the compressor keeps track of all illegal fingerprints (i.e., those fingerprints for which a collision occurred.) In the event that this state is lost (e.g., the compressor is restarted), the compressor reliably sends a cache reset message to the decompressor to ensure that the decompressor does not have any entries indexed by a previously illegal fingerprint.

Further, restarting the decompressor during operation of the protocol may result in significant rejection traffic. Therefore, we explicitly send a cache reset message from the decompressor to the compressor. This is merely a performance optimization, and is not essential for correctness.

3.4 Implementation

We implemented the architecture described above using a pair of Intel-based PentiumII 300MHz machines running Linux 2.0.31 with 128MB of RAM each. The machines were directly connected to each other via a dedicated 10 Mbps Ethernet and both machines were 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 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 Figure 1 indicates that the remaining data comprises only 3% of the replicated volume. Finally, we limited the amount of memory available for the caches, excluding the overhead induced by the hash table implementations, to 200MB each.

4 Experimental Results

To evaluate the system, we 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.

4.1 Bandwidth Reduction

Our main design goal is to reduce the amount of bandwidth consumed by replicated data. We measured bandwidth savings by inserting our system into the network at the point where we previously gathered traces; see section 2.1. 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 6 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% for the entire duration of the experiment.

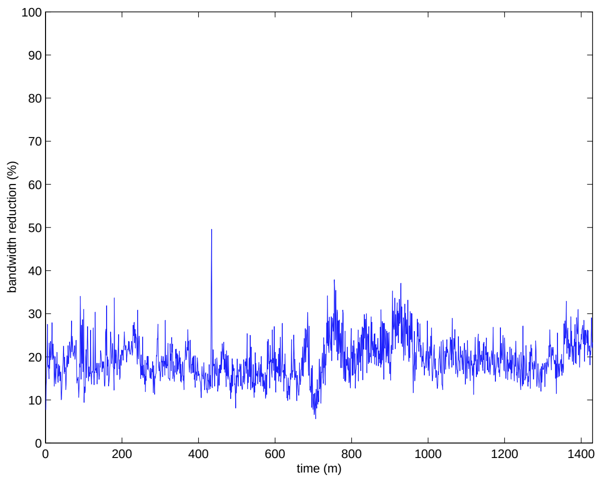

Figure 6: Bandwidth reduction for all outbound traffic

4.2 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.

Packet streams containing no detectable replication incur the highest amount of processing required for each packet at both the compressor and decompressor. This therefore presents the worst-case load for our system, and we used such streams to test the performance of the system.

To measure the throughput of the system, we ran the system over a 100 Mbps channel and sent our test stream of packets over a TCP connection that flowed over the channel. 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 use a modified Linux kernel that records 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),

Set	Reduction	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 2: Percentage of bandwidth saved by Reduction and Compression (gzip) on outbound traffic

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 an kernel-space implementation since the overhead of context switching would be removed.

4.3 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 compression to packet payloads and recorded volume statistics. To simulate useful schemes under realtime 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 2 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 less computation.

To look at the effects of combining our reduction and regular compression, we ran our trace data through a process that combined the two, first removing replicated data and compressing the remainder. Table 2 also shows these results. It highlights the fact that reduction and compression combine rather than overlap, as each tends to tap a 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 Related Work

We discuss two categories of related work: compression and caching.

5.1 Compression Techniques

When faced with a limited transfer rate, higher effective throughput can be obtained by compressing the data before transmitting it across a link. Some link protocols such as PPP make provisions for the negotiation of such a compression scheme between the ends of the link [13, 11]. Packets are then compressed (either individually or as a stream) when they enter the link and uncompressed at the other end. Alternatively, higher compression ratios can typically be obtained by compressing the data before sending it into the network. This is so for two reasons. First, lossless compression utilities such as gzip [5] work better with larger and unmixed inputs because of their statistical properties. Second, application-specific lossy schemes, such as JPEG [8] for photographic images, may be used. A further form of compression that is appropriate for some applications is delta-encoding, where a set of differences is transmitted instead of the complete object; Mogul et al. have shown that this technique may result in significant savings for Web traffic [9].

However, none of these types of compression remove the redundancy of transfers of the same information between different clients and servers whose paths cross within the network. Compression of application data can reduce the amount of information needing to be transferred, but by definition cannot remove redundancy across different clients and servers. Compression of data at the link level can in theory remove such redundancy, but in practice does not. This is because algorithms that build dynamic dictionaries typically limit their search to a small window of the data stream compared to the scale on which we will show that there is replicated traffic, e.g., gzip may search approximately 32KB, while we have detected significant correlation at 1000 times that scale.

Another type of compression that is frequently employed is packet header compression. Schemes specialized for compressing TCP/IP headers [7, 4] may reduce their impact by an order of magnitude in the best cases, and hence may have a significant impact on bandwidth usage when there are many short packets. In the packet traces we observed, however, the volume of headers was small compared with the volume of payloads, i.e., even eliminating all TCP/IP headers would not make as large a difference we demonstrated by suppressing data. Furthermore, compared to payload compression, header compression taps an orthogonal source of correlation, and could therefore be used in addition to other techniques.

5.2 Application-Level Caching

An alternative to compressing at the link level is for each application to construct its own system for caching its data-types. This is clearly not viable for all applications, but may be worthwhile in terms of bandwidth for popular cases such as the Web. To examine the tradeoffs, we briefly compare our scheme with Web caching using the generic configuration of Figure 7. 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. We ignore more costly options that reconfigure the system to shift this bottleneck, e.g., purchasing more bandwidth, spreading load over multiple links, or co-locating Web servers with the ISP.

Today's Web caches [14, 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. Our traces show

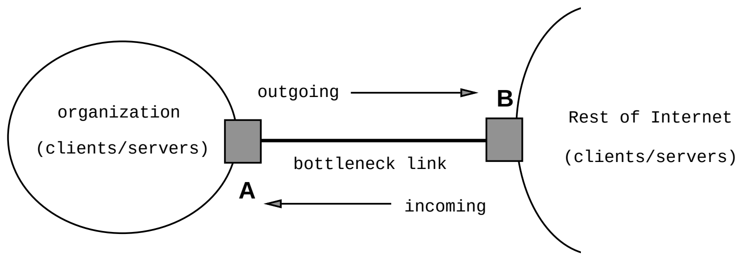

Figure 7: Generic Web Caching Configuration with a Bottleneck Link

that existing client caching has not eliminated redundant server transfers (and it is highly unlikely that this will soon be the case as it requires that all Web clients be configured in a single caching system well-matched to the underlying topology). Further, 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.

Since link layer schemes are transparent to applications, they present a different set of tradeoffs than does application-level caching. The latter may utilize application semantics, and so should be more effective for the particular application. It may also improve performance in other respects, in the same manner that Web caches lower latency as well as reduce bandwidth. However, application-level schemes may also have side-effects that a transparent scheme does not. For example, unlike Web caches, 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].

Given these tradeoffs, we believe that our scheme complements rather than competes with 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.

6 Conclusions and Further Work

In this paper, we have presented a innovative link compression protocol that suppresses the transfer of replicated 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. The protocol itself works by maintaining (nearly) synchronized packet caches at either end of a link and sending repeated payloads that are encountered as fingerprints. We have further shown by experimentation that the protocol is lightweight enough to be implemented on commodity hardware at rates exceeding T3 (45 Mbps). 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.

In addition to the bandwidth savings we realized, our scheme is significant in several respects:

- 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.

Finally, we see several areas that would benefit from further work:

- Implementation techniques such as different cache insertion and replacement policies that improve the range of match detection for a given amount of storage would improve the value of the system.
- 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.
- Additional classification techniques that increase the amount of data that we are able to detect as replicated, which would improve the effectiveness of the system as a whole.

Acknowledgments

We thank our fellow members of the Software Devices and Systems group. In particular, we wish to acknowledge Vanu Bose for insightful discussions and feedback throughout the course of this research, and John Guttag for his support and guidance. We also thank Mark Handley and Max Poletto for proofreading.

References

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

[8] I. JTC1/SC2/W10. *Digital Compression and Coding of Continuous-Tone Still Images.* IEC Draft International Standard 10918-1, 1992.

[9] J. Mogul et al. Potential benefits of delta-encoding and data compression for HTTP. In *SIGCOMM '97*, 1997.

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

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

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

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

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

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