Byte Caching in Wireless Networks

Franck Le
IBM Research
fle@us.ibm.com

Mudhakar Srivatsa
IBM Research
msrivats@us.ibm.com

Arun K. Iyengar
IBM Research
aruni@us.ibm.com

***Abstract*—The explosion of data consumption has led to a renewed interest in byte caching. With studies showing potential reductions in network traffic of 50%, this fine grained caching technique looks like a very good and attractive solution for mobile wireless operators. However, properties of wireless networks actually present new challenges. We first show that a single packet loss, re-ordering or corruption – all common conditions over the air interface – can result in circular dependencies and cause existing byte caching algorithms to loop endlessly. To remedy the problem, we then explore a new set of encoding algorithms. Third, we assess the impact of packet losses on byte caching performances, both in terms of byte savings and delay reduction. We found that a mere 1% packet loss can already nullify any delay reduction and instead cause significant increases that users may not be willing to tolerate. Finally, we shared several insights, including interactions between transport layer protocol's mechanisms (e.g., TCP window congestion) and byte caching operations that can cause sophisticated encoding algorithms to perform poorly. We believe that these insights are important for designing more efficient and robust byte caching encoding algorithms.**

I. Introduction

The growth of smartphones has caused an increase in data consumption, and brought wireless networks to their knees. Users are unable to establish data connections, and wireless networks are in dire need of solutions to alleviate network congestion. This situation has led to a regained interest in byte caching (also commonly called *data redundancy elimination*) – initially proposed by Spring and Wetherall [17] – both from the industry [4], [3], [1] and the research community [12], [6], [7], [8], [5], [18].

For a high-level description, byte caching works as follows. It relies on two main functionalities: an encoder and a decoder deployed at the end points of a resource-constrained segment. The encoder and decoder can be deployed at intermediate nodes [4], [3], [1], [6] or even at the end hosts [5], [12]. As packets are routed towards their destination, the encoder and decoder intercept the data at either the network (layer 3) or the transport (layer 4) layer of the OSI model, compute a set of fingerprints (based on a one way hash function) for each packet, and store copies of both those fingerprints and packets in a local cache. Later, if some content (or a portion thereof) is retransmitted, the encoder identifies the repeated regions, thanks to the fingerprints, substitutes them with pointers to those contents, and sends the compressed packets downstream. From the packets cached locally, the decoder reconstructs the initial packets, thus saving bandwidth and potentially lowering latency on the forwarding paths between the encoder and decoder.

Because of its simplicity, wide application to all protocols (e.g., HTTP, FTP, POP3, etc.), and fine granularity, byte caching is a very attractive solution. In contrast to HTTP caching [10], delta encoding [13] or other application specific solutions (e.g., email optimization proxies [2]), byte caching does not need to recognize nor interpret the application syntax. On the opposite, it can apply to all types of traffic (e.g., video, email, file retrieval, web browsing), and eliminates redundancy both intra-flow and inter-flows [7]. In addition, its finer granularity makes it applicable to modified content, dynamic content and media streaming. Finally, this technique applies to not only files retrieved by clients from servers, but also objects posted or uploaded by clients. Several empirical studies have been conducted and shown that byte caching can reduce network utilization by up to 50% [6], or an additional 39% saving after traditional Web proxy caching [17].

However, with a few exceptions [8], [5], most of the existing analyses have been performed on packet traces [17], [12], [6], [7], [18]. While those studies have been critical in verifying the core ideas and evaluating the potential benefits of byte caching, they present limitations.

First, they ignore important problems that can affect its proper operation. For example, Internet measurements (e.g., [9]) have established the occurrence of corruption, packet loss and reordering in the Internet routing system. Also, packet losses and corruptions are frequent events in wireless networks. We developed a complete implementation of byte caching, and demonstrate that a single occurrence of any such event (e.g., a simple packet loss) can actually result in not simply degraded performances as previously reported [12], but more importantly, circular dependencies and infinite recursions, ultimately stalling the connection(s). When byte caching is implemented on top of Transmission Control Protocol (TCP), this transport protocol handles any packet loss, re-ordering or corruption that may happen during the transmission. However, such a solution is not applicable to User Datagram Protocol (UDP) data streams. More importantly, implementing byte caching on top of TCP conflicts and causes problems with node mobility. We present the issues in Section II. On the opposite, implementing byte caching at the IP level is compatible with existing solutions (e.g., Mobile IP) and can suitably handle node mobility.

Second, trace-based studies do not allow us to analyze important metrics such as the delay between a client's input and the server's response, one of the strongest stressors in human-computer interactions. Using our complete implementation, we were instead able to evaluate the effects of packet losses on byte caching performance. While studies have reported latency gains of up to 35% [5], we found that a mere 1% packet loss can already nullify any reduction in latency. Rather than reduction, we show that packet losses, in conjunction with byte caching, can add significant delays that users may not be willing to tolerate. To briefly illustrate the reasons, we consider a sequence of n IP packets (IP_1, IP_2, ..., IP_n) and we assume that each packet, IP_k $(1 < k \leq n)$, is encoded with the previous one, i.e., IP_{k-1}. The loss of a packet, e.g., IP_1, will prevent all subsequent packets, i.e., IP_2, ..., IP_n, from being decoded. In other words, byte caching creates dependencies between IP packets which increases the resulting loss rate, and TCP performance has been shown to degrade rapidly in the presence of correlated losses as its recovery mechanisms reduce the window sizes by half and increase the timeouts exponentially. This problem was first described by Lumezanu *et al.* [12]. However, that study looked only at the impact of packet loss on bandwidth savings, not delay.

Because long delays are one of the main causes of frustration of Internet users who have been shown to bail out past a waiting threshold, we focus on the effects of byte caching on download times, especially in the presence of packet losses, packet corruption and packet re-ordering. In this paper, we make the following contributions:

- First, *we demonstrate that the existing byte caching algorithm may never terminate*. We show that a packet corruption, a packet loss or a re-ordered packet – all events which occur in the Internet – can result in cache desynchronization between the encoder and decoder, and ultimately circular dependencies which cause the encoder and decoder to loop endlessly. From the user's perspective, the TCP connections stall.
- Second, *we show that the problem is important*. We conduct a number of experiments, and more specifically, have a client retrieve a 574 KB file from a server. We observe that out of 50 file retrieval attempts, with a 1% packet loss rate, the client could successfully download the file only once. 49 out of 50 runs resulted in TCP connection stalls. With reseachers showing that 50% of the volume from Web traces have size larger than 4 MB [11], the problem could affect a large fraction of the Internet traffic.
- Third, *we propose a new set of encoding algorithms that are more robust to packet losses, packet corruption and packet re-ordering*. In particular, the proposed solutions break the circular dependencies and prevent the above problem.
- Fourth, *we evaluate the impact of byte caching on bandwidth savings and download times, in the presence of packet losses, packet corruption and packet re-ordering*. We implemented byte caching, and measured the traffic utilization and download times for varying packet loss rates. Although the newly proposed encoded algorithms offer savings in the number of bytes even with 10% packet losses, byte caching can already double the download times with even only 2% loss rate.
- Finally, *we share insights we learned from the experiments*. We found that encoding algorithms that enable high degrees of compression can surprisingly provide poorer performance. We present the observations we discovered and explain reasons behind the results. These insights are important for the design of more robust and efficient algorithms.

The rest of the paper is organized as follows. First, we provide background information on byte caching, and motivations for this study (Section II). Second, we describe further details on the byte caching algorithms, and an overview of the system we implemented (Section III). Third, we present conditions that can cause byte caching to never terminate, analyze the root cause of the problem, and conduct experiments to quantify the importance of the problem (Section IV). We find that the problem can occur frequently. Consequently, we introduce three solutions to break and prevent the identified circular dependencies (Section V). We evaluate the different encoding schemes and present the results in Section VI. Some sophisticated encoding algorithms surprisingly give poorer performance. We share the insights we discovered in Section VII. Finally, we discuss related work in Section VIII and conclude the paper with Section IX.

II. Motivations

The TCP transport protocol provides for a reliable and ordered delivery of packets. As such, one may argue that performing byte caching on TCP traffic is preferable as it directly handles any issue related due to packet loss, packet re-ordering, and packet corruption. However, such implementation presents issues with node mobility; and with smartphones and emerging tablets supporting heterogeneous network connectivity (e.g., WiFi, cellular), mobility, especially that between heterogeneous access networks (e.g., from cellular to WiFi) is becoming increasingly important for mobile wireless operators. If byte caching is done at the TCP level, this can result in disconnections for mobile clients which can interfere with data transfers. As we show below, this problem can be alleviated by performing byte caching at the IP level.

First, we observe that not only the radio interface, but also the backbone infrastructure, of cellular networks is currently under heavy stress. As such, as illustrated in Figure 1, mobile network operators are considering deploying byte caching gateways in their backbone network. However, traditional implementations of byte caching over TCP traffic present issues with node mobility (Section II-A). In contrast, performing byte caching on IP packets permits mobility (Section II-B), although such approach is sensitive to packet losses – a problem that we consequently further explore and address in this paper.

A. Byte Caching TCP traffic and Node Mobility

Byte caching is typically implemented at the TCP transport layer in a transparent manner [3], [1]. Contrary to traditional explicit proxies, this transparent mode renders the presence of the byte caching gateways invisible to the clients and

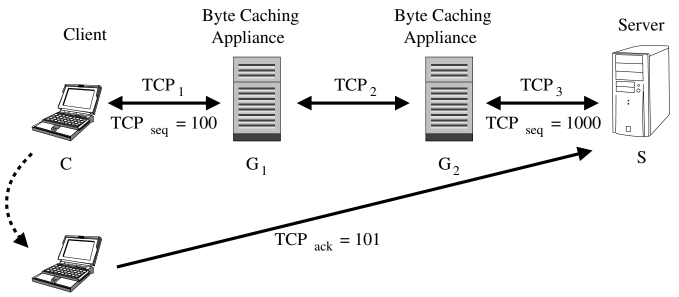

Fig. 1. Issues with mobility

eliminates any requirement for configuration at the clients. We illustrate the transparent mode and the problem such implementation creates with mobility. As depicted in Figure 1, we consider a client C retrieving an object from a HTTP server S.

t_1 The client attempts to connect with the server by sending it a TCP SYN message.

t_2 The local gateway, G_1 intercepts the SYN, and completes the TCP three-way handshake with C. At the same time, it communicates with the remote gateway G_2 to establish a new TCP connection with the intended server S, to retrieve the requested object. In other words, we have three separate TCP connections, each with its own initial TCP sequence numbers. However, the source and destination IP address fields of the packets sent by the client and server are unmodified by the gateways, rendering their presence invisible to both the client and server. The server starts sending data to G_2, which encodes and forwards it to G_1. G_1 reconstructs the initial payloads and sends them to the client.

t_3 At this time, the client may be moving, and consequently, connecting to the server through a different access network. For example, the client may be moving from a cellular access network to a WiFi one, and the new path to the server may not include any byte caching gateway. In particular, the packets may no longer traverse G_1. If the client changes IP address, solutions such as Mobile IP can conceal the change of IP address to the TCP layer by using the client's Home Address.

t_4 The client sends an acknowledgment message (e.g., TCP ACK = 100) to notify the server it has received the data and is ready to receive more.

t_5 The received segment acknowledgment does not match the one expected at the server since the sequence numbers actually belong to separate TCP connections, resulting in a TCP stall.

B. Byte Caching IP traffic and Node Mobility

Performing byte caching at the IP level preserves the end to end nature of TCP. Consequently, when the client moves to a different access network and packets get lost on-the-fly, the client can still recover the missing packets. To illustrate it, we reconsider the scenario of Figure 1 and assume the gateways perform byte caching at the IP level.

After the client sends the request, the server starts sending IP packets. The gateway G_2 intercepts the IP packets, searches for redundancy, and may compress the IP payloads. G_1 reconstructs the initial IP packets and forwards them to the client thus saving bandwidth on the link between G_2 and G_1. When the client moves to a different access network, packets may not reach the client but get lost during the procedure. However, because the end to end nature of TCP is preserved, when the client sends an acknowledgment message (e.g., TCP ACK = 100) from its new point of attachment to the server, this latter learns the amount of data the client has successfully received so far, and can retransmist the missing packets and resume the object download. The new path may include another pair of byte caching gateways (e.g., G_3, G_4) which could eliminate data redundancy between them, in a manner transparent to the users and that is compatible with node mobility.

Performing byte caching at the IP layer is therefore compatible with mobility. However, such approach is sensitive to packet loss, packet error, and packet redordering. Any such event can result in cache desynchronization and affect the byte caching operations. Although a TCP tunnel between the byte caching gateways could provide a reliable transmission channel, encapsulating TCP in TCP can result in the TCP meltdown problem. For these reasons, the subsequent sections look at the impact of unreliable IP forwarding on the correctness and performance of byte caching.

III. System Overview

We provide an overview of our implementation of byte caching. While several optimizations have been suggested to reduce the computing and memory requirements (e.g., [5]), we implemented and present the initial version proposed by Spring and Wetherall [17] since our focus is not on the server's load but the impact of packet losses, re-ordering and corruption on byte caching. We describe both the general concept of byte caching and the details of our implementation.

A. Byte Caching

For each packet, the encoder computes a set of fingerprints by sliding a w byte window over the packet and computing the Rabin fingerprint [14] of each window. To satisfy the computational and memory constraints, only the fingerprints whose last k bits are zero are retained. The corresponding packets are also stored in a local cache.

Then, when a subsequent packet, P_{new}, arrives at the encoder, its representative Rabin fingerprints are computed in a similar manner. If a fingerprint, r, falls in the range (i.e., ends with k zero bits), the encoder retrieves the corresponding stored packet, P_{stored}, from its cache. The newly arrived packet, P_{new}, and the stored packet, P_{stored}, are compared to verify the presence of a redundant region (two different strings could result in an identical Rabin fingerprint because of hash collisions), and to determine the boundaries of the

```
A. (Input) Packet $P_{new}$
B. (Redundancy Identification and Elimination Procedure)
 1: INITIALIZE pointer to beginning of $P_{new}$
 2: ADVANCE pointer by $w$ bytes
 3: while end of packet $P_{new}$ not reached do
 4:    COMPUTE Rabin fingerprint $r$ of window $w$
 5:    if ($r$ falls in range) AND ($r$ is present in local cache) then
 6:       RETRIEVE $P_{stored}$ from local cache
 7:       DETERMINE boundaries and length $len$ of repeated area surrounding $w$
 8:       if $len > 14$ then
 9:          SUBSTITUTE repeated area with encoding field
10:          MOVE pointer by ($len$ + $w$) bytes
11:       end if
12:       MOVE pointer to next byte
13:    end if
14:    MOVE pointer to next byte
15: end while
C. (Cache Update Procedure)
 1: INITIALIZE pointer to beginning of $P_{new}$
 2: ADVANCE pointer by $w$ bytes
 3: while end of packet $P_{new}$ not reached do
 4:    COMPUTE Rabin fingerprint $r$ of window $w$
 5:    if $r$ falls in range then
 6:       INSERT $r$, $P_{new}$ into local cache
 7:    end if
 8:    MOVE pointer to next byte
 9: end while
D. (Output) Encoded packet
```

Fig. 2. Encoder logic.

repeated content. The encoder then eliminates the redundant data, and instead indicates the Rabin fingerprint r, the length of the repeated area, and the offsets in the current and stored packets so the decoder can reconstruct the initial payload. The encoder also updates its cache by replacing the entry for r from P_{stored} to P_{new}. These steps are repeated until the encoder reaches the end of the payload.

B. Implementation

The encoder logic is presented in Figure 2. It consists of two main procedures. The first one identifies the repeated areas. An encoding field consists of a Rabin fingerprint (8 bytes), the offset in P_{new} (2 bytes), the offset in P_{stored} (2 bytes) and the length len (2 bytes) of the redundant sequence of bytes. As such, a repeated region is encoded only if its length is larger than 14 bytes (*lines B.8 – B.11*). The second procedure updates the local cache. Although not described (for simplicity), we highlight that in addition to the Rabin fingerprint and the payload, we also store the offset of the fingerprint in the packet. This allows the encoder to determine the boundaries of the repeated area between P_{new} and P_{stored} in a quicker manner. Finally, we set the parameters k and w to 4 and 16, respectively, to increase the effectiveness of byte caching. Small values of k and w are more effective as lower k selects a larger fraction of fingerprints and w determines the minimum width of the repeated area. However, for performance reasons, larger values may need to be selected [17], [18].

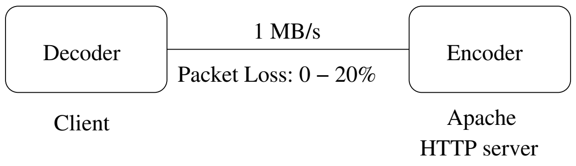

Fig. 3. Setup environment

k	**ebook**	**video**	**web page**
10	0.3%	0.009%	19–42%
100	0.7%	0.009%	26–49%
1000	1%	1%	26–52%

TABLE I
REDUNDANCY IN WEB OBJECTS

The decoder performs the reciprocal steps to reconstruct the payload.

C. Deployment Set-up

Figure 3 represents the setup deployment. A client retrieves a file from a HTTP server. We have examined our solutions on a wide range of web objects (e.g., ebooks, videos, web pages, etc.) with sizes ranging from 40KB to 6MB. To re-create the conditions of a wireless link, we rate-limit the link to 1 MBps through a traffic shaper, and we vary the packet loss rate of the link from 0 to 20%. The server is encoding the IP packets through byte caching to reduce the network utilization, and the client is supporting the decoding logic. The setup allows us to measure the impact of packet loss on byte caching. In particular, we measure both the total number of bytes sent and the download time.

We recognize that the effectiveness of byte caching depends upon the nature of web objects and temporal locality of accesses. Table I shows a baseline measurement on the effectiveness of byte caching for different types of web objects. The encoder parameter k shows the average amount of redundancy in any window of k packets. However, we note that the objective of this paper is not to examine the effectiveness of byte caching on different web objects / access patterns. Instead, our objective is to study and quantify negative interferences between byte caching algorithms and TCP retransmission schemes.

IV. TCP CONNECTION STALL

A. Problem Description

In this section we show that a naive implementation of byte caching can result in circular dependencies wherein packet P_i is encoded using packet P_j and packet P_j is encoded using packet P_i, effectively making both packets undecodable at the recipient. We also show that such circular dependencies may be triggered due to a single packet loss, packet re-order or packet corruption and can cause arbitrarily many TCP backoffs, ultimately resulting in connection termination. We trace the root cause of this problem to an interplay between the stateless nature of the byte caching algorithm and the TCP retransmission schemes.

To illustrate the problem, let us consider the following sequence of events that may occur in a naive byte caching implementation (Figure 4):

t_1 An IP packet IP_{i-1} contains a sequence m of bytes whose Rabin fingerprint r falls in the range. As such, both the fingerprint and packet are stored in the cache. The packet is then sent from the encoder to the decoder. However, because of the link characteristics, the packet is dropped and does not reach the decoder.

t_2 The subsequent IP packet IP_i contains the same sequence of bytes m with Rabin fingerprint r. As such, the sequence m is removed and replaced with an encoding field. In other words, IP packet IP_i is encoded using IP_{i-1}. The packet IP_i is then sent from the encoder and arrives at the decoder.

t_3 Upon receiving the encoded packet IP_i, the decoder attempts to reconstruct the initial payload. In particular, the encoding field indicates the fingerprint r which the decoder uses to retrieve the packet in the cache. However, because IP_{i-1} was lost, the cache has no entry corresponding to r. As such, IP_i cannot be decoded, and the packet is dropped.

t_4 Since the encoder end's TCP did not receive any acknowledgment for the data sent in IP_{i-1}, after a certain time out, it retransmits the packet. Although the packet is actually a retransmission of IP_{i-1}, at the IP layer, it appears as a new IP packet IP_{i+1}. IP_{i+1} contains the sequence of bytes m and is therefore encoded through the payload in the local cache, i.e., IP_i. The entry for r is updated with IP_{i+1}.

t_5 When receiving the encoded IP_{i+1}, the decoder attempts to reconstruct the initial payload by retrieving the entry corresponding to fingerprint r. However, the cache has no data for this key. Consequently, the packet cannot be decoded and is dropped.

The steps t_4 and t_5 keep repeating while the TCP time outs grow exponentially, and the application stops receiving data.

B. Root Cause Analysis

We note that the problem arises for the following reason: After a retransmission, a TCP segment may be encoded using, not only a preceding one, but also a succeeding one or itself. In the illustrated case (Figure 4), the first retransmition of IP_{i-1}, i.e., IP_{i+1}, is encoded using a succeeding packet, IP_i. Then, the second retransmition of IP_{i-1}, i.e., IP_{i+2}, is encoded using IP_{i+1}, i.e., itself since IP_{i-1}, IP_{i+1} and IP_{i+2} are in fact all the same TCP segment. As depicted in Figure 5, we obtain circular dependencies between the encoded packets, and the decoder cannot reconstruct the payloads.

C. Importance of the Problem

The previous sections demonstrated how a single packet loss can cause a TCP connection to stall, bringing any communication between the client and server to a halt. A natural question that ensues is: *How often does this problem happen?*

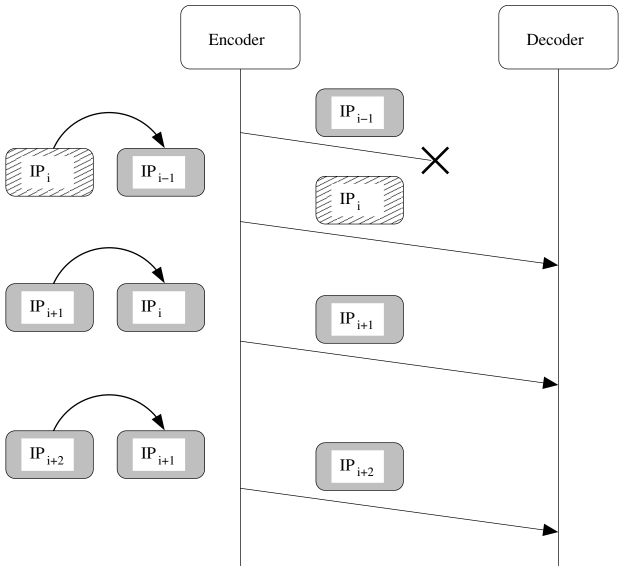

Fig. 4. Illustration of the TCP connection stall problem.

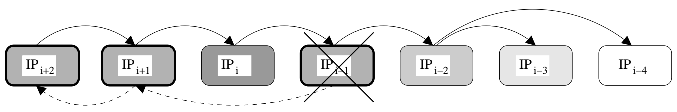

Fig. 5. Illustration of circular dependencies: As represented by the plain arrows, IP_{i+2} is encoded using IP_{i+1}, and IP_{i+1} is encoded using IP_i. In addition, as depicted by the dashed arrows, IP_{i-1}, IP_{i+1} and IP_{i+2} are in fact all the same TCP segment. Consequently, we obtain circular dependencies (e.g., IP_{i+2} – IP_{i+1} – IP_{i+2}) between the encoded packets.

To answer this question, we conducted the following experiment. We set the packet loss rate to 1%. Others such as [12] have reported much higher packet loss rates in wireless networks than the 1% we are assuming. In the presence of higher packet loss rates, the problems we have identified with byte caching will be worse.

We clear the caches at the two byte caching end points, and have the client retrieve a file from the server. The file is an e-book in text format, with a size of 587,567 bytes. We repeat the above steps 50 times, and for each run, we report the percentage of the file that has been retrieved by the client before the TCP connection stalls. Figure 6 depicts the observed results. Among the 50 attempts, only one of them successfully retrieved the file. All the others retrievals failed and got aborted prematurely because of the cyclic dependencies.

Analyzing the traces, we noticed that as soon as a packet is lost, the TCP connection stalls for the following reason: As the IP packets traverse the byte caching encoder, they are stored in the encoder's local cache. Then, when a packet is dropped on its way to the client, the packet does not reach the byte caching decoder. Upon detecting a missing segment, the TCP transport protocol retransmits the missing data. The byte caching encoder compresses the newly retransmitted segment using the previously stored IP packet. However, since the

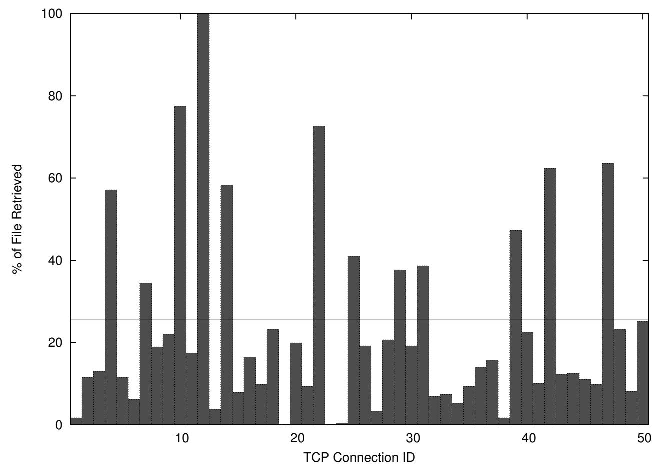

Fig. 6. Frequency of TCP connection stalls (1% Packet Loss).

required IP packet is not present in the decoder's cache, the decoder is unable to reconstruct the initial segment. The packet is dropped, and we obtain the cyclic dependencies previously described. In summary, as soon as a first packet is lost, the file retrieval comes to an end.

The horizontal solid line in Figure 6 represents the average, i.e., 25.5% (or 149,829 bytes) of the file being downloaded across the runs. With a TCP Maximum Segment Size (MSS) of 1460 bytes on Ethernet, we note that the average closely corresponds 100 IP packets, the reciprocal of the 1% packet loss rate. In other words, with a packet loss rate of 1%, approximately 146,000 bytes can on average be retrieved before the TCP connection stalls.

We note that Gill *et al.* [11] have analyzed traces of HTTP activity from a large enterprise and from a large university, and revealed that 50% of the data volume comes from objects larger than 4 MB in the enterprise trace, and 28 MB in the university trace, all sizes that are significantly larger than the 600 KB we have considered in the experiments above.

So far, we have focused on the impact of packet loss on a single TCP connection. In reality, a packet loss may cause the desynchronization between the encoder's and decoder's caches, and, not only one TCP connection, but all subsequent connections going through the encoder and decoder may get affected. As such, the problem is even more important.

V. Solutions

We explore three solutions to break the circular dependencies previously described. The first solution proposes a simple approach to break the circular dependencies. The second algorithm is more sophisticated, and attempts to achieve a higher degree of compression. Finally, the third approach is inspired from video encoding techniques, and is applicable to not only TCP but also UDP traffic.

A. Cache Flush Encoding

This first solution consists in flushing the cache upon detecting a TCP retransmission. This ensures that all retransmitted segments are encoded using neither a succeeding segment nor itself. Instead, all retransmitted segments are sent not encoded. To detect TCP retransmissions, we keep track of the TCP sequence number of outgoing segments. Any observed decrease in the TCP sequence number of an outgoing segment triggers a cache flush.

```
A. (Input) Packet $P_{new}$
B. (Redundancy Identification and Elimination Procedure)
 1: INITIALIZE pointer to beginning of $P_{new}$
 2: ADVANCE pointer by $w$ bytes
 3: **while** end of packet $P_{new}$ not reached **do**
 4:    COMPUTE Rabin fingerprint $r$ of window $w$
 5:    **if** ($r$ falls in range) AND ($r$ is present in local cache) **then**
 6:       RETRIEVE $P_{stored}$, $TCP\_seq_{stored}$ from local cache
 7:       **if** $TCP\_seq_{new} > TCP\_seq_{stored}$ **then**
 8:          DETERMINE boundaries and length $len$ of repeated area surrounding $w$
 9:          **if** $len > 14$ **then**
10:             SUBSTITUTE repeated area with encoding field
11:             MOVE pointer by ($len$ + $w$) bytes
12:          **end if**
13:          MOVE pointer to next byte
14:       **end if**
15:       MOVE pointer to next byte
16:    **end if**
17:    MOVE pointer to next byte
18: **end while**
C. (Cache Update Procedure)
 1: INITIALIZE pointer to beginning of $P_{new}$
 2: ADVANCE pointer by $w$ bytes
 3: **while** end of packet $P_{new}$ not reached **do**
 4:    COMPUTE Rabin fingerprint $r$ of window $w$
 5:    **if** $r$ falls in range **then**
 6:       INSERT $r$, $P_{new}$, $TCP\_seq_{new}$ into local cache
 7:    **end if**
 8:    MOVE pointer to next byte
 9: **end while**
D. (Output) Encoded packet
```

Fig. 7. TCP Sequence Number encoding algorithm.

B. TCP Sequence Number Encoding

Although the previous solution addresses the problem, it may be too drastic. First, it prevents all retransmitted TCP segments from being encoded. Although, we indeed want to avoid a TCP segment from being encoded using a succeeding segment or itself, a retransmitted TCP segment can still be encoded using a preceeding segment. Second, by flushing the cache, the redundancy elimination of the TCP segments following a TCP retransmission is also affected as the cache history is shortened.

As such, we propose the encoding algorithm presented in Figure 7. When a fingerprint falls in the range, we store not only the packet, the offset of the fingerprint in the payload, but also the TCP sequence number (*line C.6*). In addition, for the redundancy identification and elimination procedure, we encode a repeated region only if it is present in a preceding

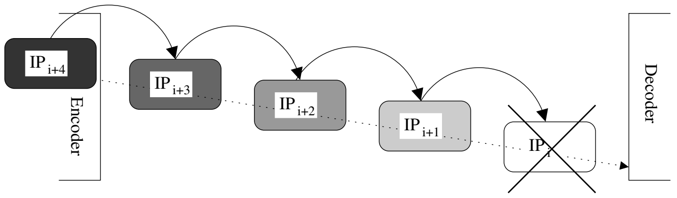

Fig. 8. An entire window of packets may already be sent and encoded before the encoder detects a packet loss.

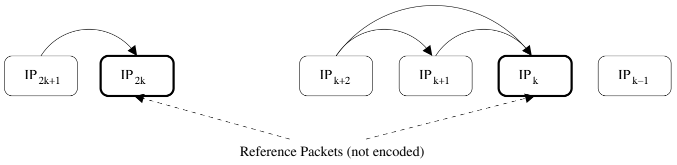

Fig. 9. k-distance for reducing cascading dependencies on lost packets. A packet can only be encoded using the immediately preceding reference, and any of the previous packets until that reference. In this figure, packets IP_k, and IP_{2k} are references. Packet IP_{k+2} can be encoded using only IP_{k+1}, and IP_k.

TCP segment. We enforce it by verifying that the TCP sequence number of the stored packet is strictly lower than the current packet (*line B.7*). Otherwise, the redundant area is left not encoded.

The proposed logic ensures that a TCP segment is not encoded using a succeeding segment or itself but enables it to be encoded using any preceeding segment. Compared to the previous solution, this approach does not require the cache to be flushed.

C. *k-distance Encoding*

The third proposal is motivated by the observation that to more efficiently use the available bandwidth, and increase the throughput, TCP allows a full window of data to be in-transit. The TCP window scale option, as defined in RFC 1323, actually increases the TCP window size from 65,535 bytes to even 1 Gigabyte.

Consequently, even with the previous scheme, a long sequence of packets may already be sent before the encoder detects a packet loss. As depicted in Figure 8, all the packets in transit will not be decodable but will have to be re-transmitted.

To limit this effect, we propose the following encoding algorithm, inspired from the MPEG-4 compression system. Every k packets, we send a *reference* not encoded, i.e., a packet not encoded using any other datagram. The subsequent $k-1$ packets can be encoded using the immediately preceding reference, and any of the previous IP packets until that reference (see Figure 9). This scheme ensures that no more than k packets are dropped ensuing a packet loss. We call k, the *distance*.

VI. Evaluation

We evaluated the performances of the three proposed solutions, and compared them with those observed when no byte caching is applied. We repeated the following experiments for

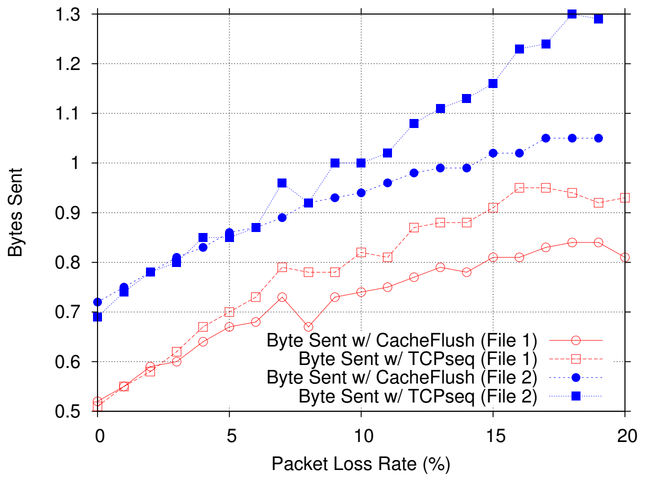

Fig. 10. Byte savings in the presence of packet losses.

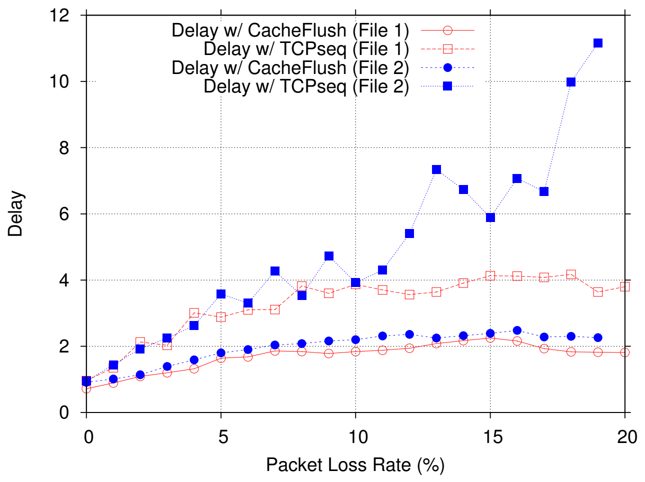

Fig. 11. Download times in the presence of packet losses.

a number of objects but only present the most representative results.

First, we varied the packet loss rates and measured both the byte savings (Figure 10) and download times (Figure 11) for the Cache Flush and TCP Sequence Number encoding algorithms. In both figures, the y-axis represents the ratio:

$$\frac{Number\ of\ sent\ bytes\ (respectively,\ delay)\ when\ DRE\ is\ applied}{Number\ of\ sent\ bytes\ (respectively,\ delay)\ without\ DRE}$$

In the absence of packet loss, data redundancy elimination can reduce the number of sent bytes by 45% and the download time by 28%. These results are consistent with previous studies [7], [5].

However, although the new encoding algorithms are more robust to packet losses and can offer byte savings even with 10% packet loss, the presence of even only 1% packet loss can already nullify any delay reduction and actually increase the delay by up to 35%; and 2% packet loss rate can even multiply the delay by a factor of 2.

We represent the results of the algorithms applied to two different files. File 1 has an average number of dependencies

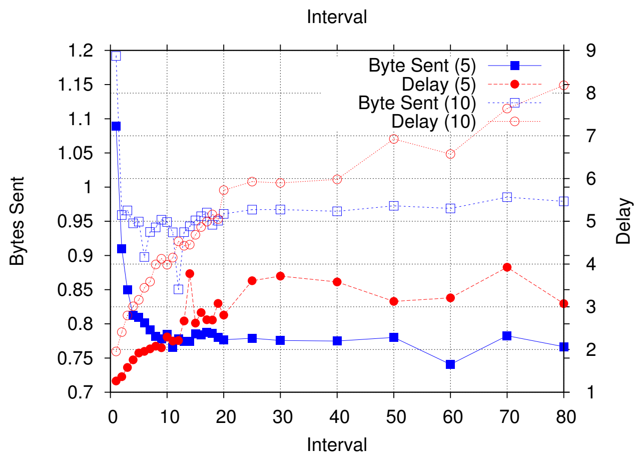

Fig. 12. Performance of k-distance algorithm for 5% and 10% packet loss for File 1. The bytes sent axis is normalized by the file size, while the delay is normalized by the download times in the absence of packet losses. Finally, the x-axis represents the distance k.

to distinct IP packets of 4 while file 2 has an average of 7. We observe that file 2 is more sensitive to packet losses both in terms of byte savings and delay. This confirms our hypothesis: When the number of dependencies is larger, packet losses can have a larger impact on the performance as they induce more correlated losses.

When comparing the results of the encoding schemes between them, Cache Flush is generally performing better than TCP Sequence Number both from a byte saving and download time perspective. These observations may be surprising, given that the latter scheme should allow more redundant data to be eliminated. We further investigated to understand the rationale behind those results, and present the findings in the next section.

Second, we evaluate the performance of the k-distance encoding algorithm (see Figure 12). We varied the distance k and measured both the sent bytes, and delay for a packet loss rate of 5% and 10%. A value for k of approximately 8 provides a reasonable tradeoff between sent bytes versus delay. In particular, it offers a 24% byte savings while still limiting the delay. Table II compares the results with those of the Cache Flush and TCP sequence algorithms.

We note that even when varying the k distance from 0 to 80, we cannot achieve the gains offered by Cache Flush. For example, considering a 5% packet loss rate, the k-distance algorithm always provides less than 30% byte savings even when increasing the value of k to 80. In contrast, at the same loss rate, we note that Cache Flush can offer more than 30% byte savings. This further motivates the reasons to investigate why Cache Flush is counter intuitively performing better than the two other more sophisticated schemes.

VII. Insights: Ineffectiveness of Aggressive Compression

We proposed three encoding algorithms in Section V: Cache Flush, TCP sequence number, and k-distance. The two latter ones are more sophisticated and were expected to give better performance. For example, the TCP sequence number algorithm allows the encoder to not only eliminate all the redundancy identified by Cache Flush, but also to identify additional repeated content(s) with previous packets. As such, the TCP sequence number algorithm was expected to provide higher bandwidth savings, and hence potentially lower latency, than Cache Flush. However, the experiments conducted in Section VI showed the opposite: the simpler Cache Flush algorithm surprisingly offers higher byte savings and lower latency.

	Cache Flush	TCP seq	k-distance
Bytes Sent (5% loss)	0.67	0.70	0.76
Delay (5% loss)	1.64	2.88	2.11
Bytes Sent (10% loss)	0.74	0.82	0.94
Delay (10% loss)	1.84	3.87	4.01

TABLE II
Comparison of all three encoding scheme for File 1, and packet loss rates of 5% and 10%. For the k-distance algorithm, we set k to 8.

We measured the perceived packet loss rate due to different algorithms, where perceived loss rate denotes the aggregate loss rate due to the communication channel and due to the fact that certain packets are undecodable (i.e., IP_i depends on IP_j, but IP_j is lost by the communication channel). Figure 13 shows the perceived packet loss rate against the actual packet loss rate due to the underlying communication channel.

First, we observed that the perceived packet loss rate for the TCP sequence algorithm is significantly higher than that of the Cache Flush algorithm. We conjecture that this results because the TCP sequence algorithm is more aggressive in compressing packets and causes more complex inter-packet dependencies which in turn impacts the perceived packet loss rate. Consequently, the benefits due to more aggressive compression are being offset due to higher perceived packet loss rate and subsequent TCP retransmissions.

To confirm the above conjecture and to understand the reasons behind the higher perceived loss rates, we delved further into the dependencies between the encoded packets. We deployed the TCP sequence number algorithm between the client and server. Figure 14 illustrates an extract of the packets exchanged between them. The upper arrows represent the dependencies: e.g., IP_{14} includes redundant content with, and is therefore encoded using IP_{13} and IP_{12}. The lower arrows highlight the retransmissions: e.g., IP_{24} is a retransmission of IP_{13}. We observed the following sequence of events:

t_1	IP_{13} is sent from the server to the client. However, it is randomly dropped and does not reach the client.
t_2	IP_{14} is encoded using IP_{13}. Although IP_{14} is properly received by the client, it cannot be decoded as IP_{13} is missing. IP_{14} is therefore dropped.
t_3	Similarly, IP_{15} to IP_{23} have some either direct or indirect dependencies to IP_{13}, and cannot be decoded. They are all dropped by the client.

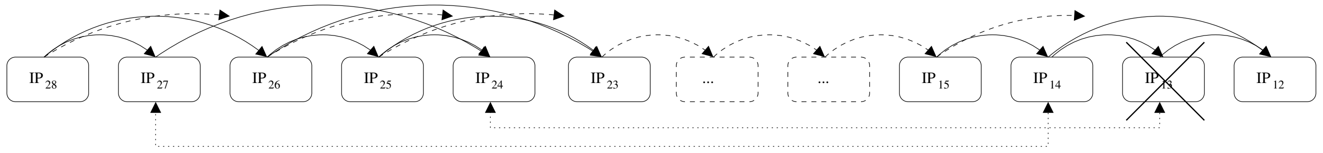

Fig. 14. Dependencies between IP packets.

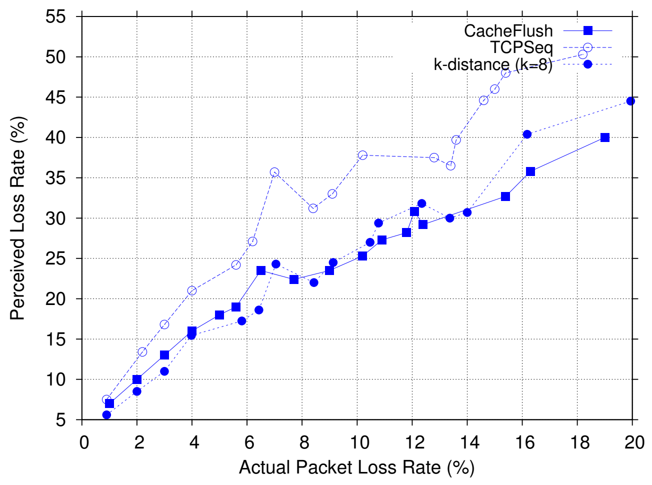

Fig. 13. Perceived Packet Loss Rate

t_4 IP_{24} is a retransmission of IP_{14}. It is sent not encoded. The client accepts it and forwards it to the upper layer, i.e., TCP.

t_5 IP_{25} is encoded using IP_{24}, IP_{23} and IP_{21}. Since IP_{23} and IP_{21} are missing at the client, IP_{25} cannot be decoded but is dropped.

t_6 IP_{26} is encoded using IP_{25}, IP_{23} and IP_{21}. As the previous packet, IP_{26} cannot be decoded.

t_7 IP_{27} is a retransmission of IP_{14}. It is encoded using IP_{24}. The packet is properly decoded.

t_8 IP_{28} is encoded using IP_{27}, IP_{26} and IP_{20}. IP_{28} cannot be decoded as the client has neither IP_{26} nor IP_{20}.

Among all the packets IP_{14} to IP_{28} sent from the server to the client, following a packet loss (IP_{13}), only two packets (IP_{24} and IP_{27}) were properly received and forwarded to the TCP layer. In other words, an entire window of packets was already sent and encoded using the loss packet, and subsequent packets may be further encoded using any of these already sent datagrams. Consequently, the impacts of a lost packet are seen even after an entire window of packets.

To highlight the differences and reasons Cache Flush offers better results, let us assume that Cache Flush is deployed between the client and server: a cache flush would happen before IP_{24} is sent. Then at t_5, IP_{25} would be encoded using only IP_{24}. As such, the receiver could correctly decode IP_{25}, resulting in a lower perceived loss rate. In summary, Cache Flush limits the dependencies to a window of packets.

Second, we focus on the perceived packet loss rate for the k-distance algorithm, and observe that the results are very similar to those of Cache Flush (Figure 13). One may therefore expect these two encoding algorithms to give comparable performances. However, Section VI showed that Cache Flush was actually offering higher savings and lower latencies. It turns out that for small values of k, the k-distance algorithm forgoes opportunities to compress by limiting the number and the choice of packet encodings to a window of at most k packets. For instance, when $k = 8$ and $p = 9\%$, our experiments showed that the average packet sizes for the cache flush algorithm and the k-distance algorithm were 835 bytes and 920 bytes respectively (while the numbers of packets sent by both the algorithms were nearly identical, around 390 packets). As such, although the perceived loss rate is comparable, for small values of k, the average packet size of the k-distance algorithm is larger than that obtained with Cache Flush, explaining the differences in performances. As we increase k ($> \frac{1}{p}$), the perceived packet loss rate of the k-distance algorithm evidently surpasses the cache flush algorithm. Intuitively, for large values of k, the behavior of the k-distance algorithm must match that of the TCP sequence number algorithm. For instance, when $k = 50$ and $p = 9\%$, our experiments showed that the average packet size for the k-distance algorithm drops to 634 bytes, while the total number of packets sent by the k-distance algorithm increases to 430 – indicating that more aggressive compression may come at the cost of higher perceived packet loss rate, which in turn offsets the benefits due to compression.

We believe that our findings point to an important cross-layer optimization problem (IP layer mobility and redundancy elimination vs. TCP layer retransmissions). When implementing byte caching at the IP layer, special attention must be paid to IP layer byte caching so as to not interfere with TCP layer retransmissions. We believe that our initial study warrants a more detailed investigation into such cross-layer problems with the goal of optimizing end-to-end network performance.

VIII. Related Work

Caching of frequently requested data objects has been used for reducing bytes transferred in many network systems, particularly for the Web [15]. However, caching of complete objects cannot take advantage of partial redundancies among similar but not exact objects. Fragment-based Web caching has been used for reducing redundant storage and transmission of Web pages which have similar but not identical content [16].

In order to address some of the limitations of Web caching, Spring and Wetherall [17] proposed a protocol-independent algorithm for reducing redundant network traffic. For the Web traces used in this paper, even after proxy caching was applied, an additional 39% of the traffic was found to be redundant.

The redundancy elimination algorithm that we used in this paper is based on the one proposed by Spring and Wetherall.

Anand *et al.* have worked extensively on byte caching [6], [7], [8], [5]. A key difference in our work is that we consider the effect of packet losses. We demonstrate that unreliable packet delivery requires enhancements to the byte caching algorithm for correctness and also has a significant effect on performance.

Lumezanu *et al.* examined the effect of packet losses on redundancy elimination in cellular wireless networks [12]. They studied Spring and Wetherall's algorithm and found that the loss of only a few packets can disrupt redundancy elimination and eliminate bandwidth savings. They proposed an approach called informed marking to detect lost packets and prevent them from being used for future encodings. Our work confirms that packet losses can significantly reduce the value of redundancy elimination. We go beyond [12] by pointing out correctness problems in the presence of packet losses, and quantify the effect of packet losses on download times, something which is not done in [12].

Finally, we discuss two additional potential approaches to address packet losses: a first solution could consist in having the decoder – upon detecting a missing packet – sending a notification message to the encoder to retrieve a copy of the missing actual content. We have not evaluated and are planning to assess the impact of packet losses on the performances of such a solution. However, we speculate that the extra round trip required to retrieve the missing packet can still result in a large number of dependencies affected by the loss, and hence, cause considerable delays in TCP performances. A second solution could consist in not caching a packet until it has been successfully acknowledged as received by the other endpoint. However, the lost of an acknowledgment message could still lead to cache desynchronization.

IX. Conclusion

This paper presents an in-depth implementation based analysis of IP-layer byte caching schemes. While most past studies have claimed significant gains due to byte caching schemes we show: (i) a naive implementation of IP-layer byte caching scheme may violate correctness requirements by introducing circular dependencies which may be triggered even due to a single packet loss (or packet corruption or reordering) and result in the termination of the TCP connection, (ii) we show that more aggressive compression mechanisms for IP-layer byte caching may negatively interfere with TCP retransmissions in the presence of packet losses; in particular, this paper shows that the savings due to more aggressive compression mechanisms may be offset due to higher perceived packet loss rates and TCP retransmissions, and (iii) unlike past trace based studies our implementation based study shows that while byte savings may be achieved even in the presence of packet losses, the impact of byte caching on latency may be non-trivial (e.g., a factor of 2 increase in latency at 2% packet loss).

Indeed one potential solution to circumvent such problems is to implement byte caching schemes over the TCP layer; however, such schemes may not seamlessly support IP mobility or may incur the overhead of TCP state migration. We believe that this paper improves the design of efficient and robust byte caching schemes by highlighting: (i) possible negative interferences that arise due to cross-layer optimization (in particular, between TCP retransmission schemes and IP-layer byte caching schemes) and (ii) the need to build a tuneable byte caching scheme that can dynamically adapt how aggressively it compresses packets based on the packet loss rate in the underlying communication channel.

X. Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-09-2-0053. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] Blue Coat. http://www.bluecoat.com/.
[2] Bytemobile. http://www.bytemobile.com/.
[3] Cisco Wide Area Application Services. http://www.cisco.com/go/waas.
[4] Riverbed Technology. http://www.riverbed.com/.
[5] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. EndRE: An End-system Redundancy Elimination Service for Enterprises. In *NSDI*, 2010.
[6] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet Caches on Routers: The Implications of Universal Redundant Traffic Elimination. In *SIGCOMM*, 2008.
[7] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee. Redundancy in Network Traffic: Findings and Implications. In *SIGMETRICS*, 2009.
[8] A. Anand, V. Sekar, and A. Akella. SmartRE: An Architecture for Coordinated Network-wide Redundancy Elimination. In *SIGCOMM*, 2009.
[9] J. Bellardo and S. Savage. Measuring Packet Reordering. In *SIGCOMM Workshop on Internet Measurement*, 2002.
[10] I. Cooper, I. Melve, and G. Tomlinson. Internet Web Replication and Caching Taxonomy. RFC 3040, 2001.
[11] P. Gill, M. Arlitt, N. Carlsson, A. Mahanti, and C. Williamson. Characterizing Organizational Use of Web-based Services: Methodology, Challenges, Observations, and Insights. In *ACM Transactions on the Web (TWEB)*, 2011.
[12] C. Lumezanu, K. Guo, N. Spring, and B. Bhattacharjee. The Effect of Packet Loss on Redundancy Elimination in Cellular Wireless Networks. In *IMC*, 2010.
[13] J. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. van Hoff, and D. Hellerstein. Delta Encoding in HTTP. RFC 3229, 2002.
[14] M. O. Rabin. Fingerprinting by random polynomials. In *Technical Report TR-15-81, Department of Computer Science, Harvard University*, 1981.
[15] M. Rabinovich and O. Spatscheck. *Web Caching and Replication*. Addison-Wesley, 2001.
[16] L. Ramaswamy, A. Iyengar, L. Liu, and F. Douglis. Automatic fragment detection in dynamic web pages and its impact on caching. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):859–874, June 2005.
[17] N. Spring and D. Wetherall. A protocol-independent Technique for Eliminating Redundant Network Traffic. In *ACM SIGCOMM*, 2000.
[18] K. Tangwongsan, D. G. Andersen, M. Kaminsky, and H. Pucha. Efficient Similarity Estimation for Systems Exploiting Data Redundancy. In *INFOCOM*, 2010.