Cooperative End-to-End Traffic Redundancy Elimination for Reducing Cloud Bandwidth Cost

Lei Yu[†], Karan Sapra[†], Haiying Shen[†] and Lin Ye[‡]
[†]Department of Electrical and Computer Engineering, Clemson University, Clemson, SC, USA
[‡]Department of Computer Science and Technology, Harbin Institute of Technology, China
{leiy, ksapra shenh}@clemson.edu, yelin@nis.hit.edu.cn

***Abstract*—The pay-as-you-go service model impels cloud customers to reduce the usage cost of bandwidth. Traffic Redundancy Elimination (TRE) has been shown to be an effective solution for reducing bandwidth costs, and has recently captured significant attention in the cloud environment. By studying the TRE techniques with a trace driven approach, we found that solely using either sender-based TRE or receiver-based TRE cannot simultaneously capture traffic redundancy in both short-term (time span of seconds) and long-term (time span of hours or days) data redundancy, which concurrently appear in the traffic. Additionally, the TRE efficiency of existing receiver-based TRE solution is susceptible to data changes compared to historical data in the cache. In this paper, we propose a sender and receiver Cooperative end-to-end TRE solution (CoRE) for efficiently identifying and removing both short-term and long-term redundancy. Through a two-layer redundancy detection design and one single pass algorithm for chunking and fingerprinting, CoRE efficiently carries out cooperative operations between the sender and the receiver. By extensive evaluation with several real traces, we show that CoRE is able to identify both short-term and long-term redundancy with low additional cost, while ensuring TRE efficiency from data changes.**

I. Introduction

Cloud computing is an emerging IT paradigm that provides utility computing by a pay-as-you-go service model [1]. More and more organizations are moving their data and services to the cloud, which accordingly drive increased bandwidth demands and costs for data access. The cost of cloud hosting services to these organizations can vastly increase due to the usage-based bandwidth pricing. Thus, the bandwidth cost became an essential concern in cloud and is receiving great attentions [2], [3].

In order to reduce bandwidth cost of data transfer from the cloud, Traffic Redundancy Elimination (TRE) technologies have being exploited [4]. Since significant redundancy has been found in the network traffic [5], [6], due to common accesses to the same or similar data objects from the Internet end-users, eliminating the transmission of duplicate information can greatly reduce the bandwidth usage. A number of TRE solutions have been proposed for WAN optimization, including pair middlebox-based solution placed at either end of a WAN link [7], [8], [9], and end-to-end solution EndRE [10] deployed at client and server side. They maintain fully synchronized caches at both sender and receiver sides. The sender detects the duplicate contents by comparing the outgoing data with its local cache and sends the reference of duplicate data to the receiver instead of raw data. However, it has been shown that they are deficient for a cloud environment because of the elasticity and usage-based pricing of the cloud [4]. First, the workload distribution and migration due to cloud elasticity can lead to frequent changes of service points for the clients. Both the pair middlebox-based solution and EndRE require tight cache synchronization between two end sides, which is difficult or costly to maintain in such dynamic environment. Second, the deployment of TRE solution can incur additional usage of computation and storage resources at cloud servers. Without careful design and implementation, the costs of running TRE maybe eradicate the bandwidth cost savings provided by TRE. EndRE, a sender-based TRE solution which offloads most processing effort and memory cost to servers, suffers such risk [4].

Recently, a receiver-based TRE solution named PACK [4] is proposed to address the above issues arising in cloud environment. In PACK, once a client receives a data chunk that already exists in its local cache, it is expected that the future coming data are also matched with its cached data. The client makes predictions for future coming data chunks and notifies the cloud server. The server confirms the correctly predicted chunks, which then do not need to be transferred. Without maintaining client status at the servers, PACK effectively handles cloud elasticity. By predicting future traffic redundancy at clients, PACK offloads most computation and storage cost from cloud to clients, and thus greatly reduce the TRE cost in cloud.

Recent studies on network traffic redundancy [6] have shown that vast majority of data matches with the cache are for chunks of size less than 150 bytes and have high degree of temporal locality such as 60% within 100 seconds, and popular chunks can recur with the time difference as large as 24 hours. Such results indeed indicate two types of traffic redundancy, referred to as *short-term traffic redundancy* (repetition in minutes) and *long-term traffic redundancy* (repetition in hours or days) according to the time scale of repetition occurrence. By studying real traces we collected, we demonstrate that the short-term and long-term redundancy can concurrently appear in the network traffic. PACK can effectively capture long-term redundancy in traffic between a server and a client, because it only requires data caching at clients which can be on large-size persistent storage such as disks and kept for a long term. In contrast, sender-based solution EndRE is not suitable for exploiting long-term traffic redundancy. EndRE needs to maintain a large-size cache at the server for each client to capture the long-term traffic redundancy between them. This is not feasible because the server often serves a large amount of cloud users but it only has limited storage. Nevertheless, PACK fails to capture short-term redundancy, because it uses an average chunk size of 8KB and cannot detect the short-term redundancy that appears at fine-granularity (e.g., the 150 bytes). Since a large portion of redundancy is found in short-term time scale, PACK cannot exploit the full redundancy in the network traffic. By reducing chunk size, PACK may detect fine-granularity short-term redundancy, but it will greatly increase the number of chunks and result in prohibitive prediction transmissions.

In this paper, we aim to design a TRE solution for the cloud environment, with the goals to remove the traffic redundancy as much as possible while still catering to the characteristics of the cloud. By exploiting both short-term and long-term redundancy, traffic redundancy can be eliminated to the highest degree. However, it is challenging to simultaneously and effectively remove short-term and long-term redundancy while avoiding tight cache synchronization and accumulative TRE cost for cloud application. To address this problem, we propose a sender & receiver Cooperative end-to-end TRE solution, namely CoRE, which involves two layers of TRE operations at the sender and can adaptively distribute the TRE effort between the sender and receiver. With efficient joint efforts of sender and receiver, CoRE is able to ensure high TRE efficiency with low additional cost at cloud servers and clients.

In CoRE, the first-layer TRE performs prediction-based *Chunk-Match*. To ensure prediction efficiency against data changes, we improve prediction-based TRE compared with PACK. Current prediction-based design in PACK requires that the matched chunk exactly appears at the expected position in TCP stream. Even a small offset of the outgoing data at the sender, compared with the data cached at the receiver, can invalidate the predictions and greatly degrade TRE efficiency. Thus, the sender in CoRE divides the data into chunks and compares the signatures of chunks with all recently received predictions regardless of the predicted positions. In this way, CoRE achieves resiliency against data offset and maximizes the use of predictions from the receiver. The second-layer TRE identifies maximal duplicate regions among chunks within a temporary local cache, referred to as *In-Chunk Max-Match*. Once the redundancy detection at the first-layer fails, CoRE turns to the second-layer to identify finer-granularity redundancy within chunks to gain as much bandwidth savings as possible. In this way, both long-term traffic redundancy and short-term traffic redundancy are detected.

In summary, the contributions of this paper are described as follows:

1. By a real trace driven study, we identify the limitations of existing end-to-end TRE solutions for simultaneously capturing short-term and long-term redundancy of data traffic.

2. We propose a sender & receiver cooperative TRE scheme (CoRE). To efficiently carry out cooperative operations, we design a two-layer TRE scheme. We also propose an efficient one pass fingerprinting and chunking algorithm.

3. We improve the design of prediction-based approach for ensuring TRE efficiency from data changes.

4. We implement CoRE and quantify its benefits and costs based on extensive experiments using several network traffic traces.

The rest of the paper is organized as follows. Section II describes existing TRE solutions. Section III discusses TRE solution for cloud and its limitations. Section IV presents our collaborative end-to-end TRE solution, CoRE, in detail. Section V presents our implementation. In Section VI, we evaluate CoRE and compare it with PACK by extensive experiments using several traffic traces.

II. RELATED WORK

Recently, several TRE techniques have been proposed to suppress duplicate data from the network transfers. A protocol-independent packet-level TRE solution was first proposed in [5]. The sender stores recently transferred packets, computes the *Rabin fingerprints* [11] for each packet by applying a hash function to each 64 byte sub-string of the packet content, and selects a subset of representative fingerprints as to the packet content. For an outgoing packet, the sender checks whether its representative fingerprints have appeared in earlier stored packets. If yes, the sender identifies the maximal overlap region around every matched fingerprint and replaces the region with a fixed-size pointer into the cache. To decode compressed data, the receiver replaces the pointer by the corresponding referred data in its local cache which stores recently received packets. Several commercial vendors have developed such protocol-independent TRE algorithms into their "WAN optimization" middle-boxes [7], [8], [9]. The successful deployment of TRE solutions in enterprise networks motivated the exploration of TRE deployment at routers across the entire Internet and redundancy-aware routing [12], [13].

Recent studies on traffic redundancy [6] found that over 75% of redundancy were from intra-host traffic, which implies that an end to end solution is very feasible for redundancy elimination. Accordingly a sender-based end-to-end TRE named EndRE was proposed for enterprise networks [10]. By maintaining a fully synchronized cache for each client at server, EndRE offloads most processing effort and memory cost to servers and leave the client only simple pointer lookup operations.

Most recently, a receiver-based end-to-end TRE PACK is proposed for cloud environment [4]. The receiver divides the incoming data stream into chunks, links and stores them in sequence, referred to as a *chain*. The receiver compares each incoming chunk to its local chunk store. Once finding a matching chunk, it retrieves a number of subsequent chunks on the chain where the matching chunk is located. The signatures of these chunks and their expected offsets in the incoming data stream are sent in a PRED message to the sender as a prediction for the sender's subsequent outgoing data. Figure 1 briefly describes the PACK algorithm. Once finding a match with an incoming chunk [*XYZA*] in the chunk store, the receiver sends to the sender the triples of signature, expected

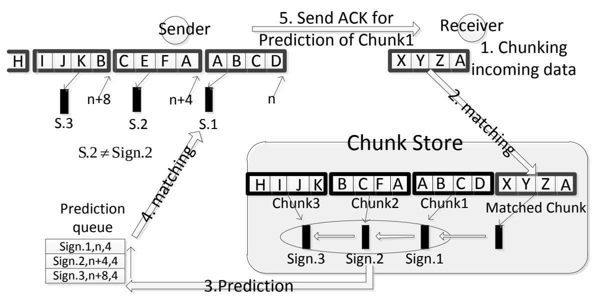


Fig. 1. A brief description for PACK algorithm. *s.#* and *sign.#* are chunk signatures. $n + \#$ is the expected offset of a predicted chunk in TCP stream.

offset and length of following chunks $[ABCD]$, $[BCFA]$ and $[HIJK]$ in the same chain, i.e., $(Sign.1, n, 4)$, $(Sign.2, n+4, 4)$ and $(Sign.3, n+8, 4)$. According to these predictions, the sender computes SHA-1 over the outgoing data with the expected offset and length, and compares the result with the corresponding prediction. Upon a signature match, the sender sends a PRED-ACK message to the receiver to tell it to copy the matched data from its local storage. The data that is obtained by the receivers from local cache instead of real transmission is referred to as *virtual data*. PACK introduces *adaptive virtual window* to limit the maximum total size of chunks that can be predicted each time, thereby adjusting the receiving rate of virtual data dynamically.

III. TRE SOLUTIONS FOR CLOUD AND LIMITATIONS

A. Impact of Short-term and Long-term Traffic Redundancy

In this section, we conduct trace-driven study to show the impact of short-term and long-term traffic redundancy on the efficiency of current end-to-end TRE solutions, and identify their limitations.

1) Short-term redundancy: The real Internet traffic trace we used is captured from an access link from a large university to the backbone. The trace is 120 seconds long and contains 1.9GB HTTP traffic between 8277 host pairs including payloads. The detailed description is given in Table I. For every host pair of sender and receiver, we detect redundancy in the traffic from server to client by PACK and a sender-based TRE similar to EndRE's Max-Match [10] respectively. The experimental results show that PACK has detected little redundancy, totaling 1.4MB in 1.9GB traffic. By contrast, our results show that the sender-based TRE using a small cache size of 250KB has detected 5% redundancy, which amounts to 114MB.

Figure 2 shows the length distribution of matched data found by the sender-based TRE in our traffic trace. We can see that about 70% of the matches have size no more 150 bytes. The results indicate that data repetition occurs within short-term time scale less than 120s and mostly with size at the order of hundred bytes. The large difference on TRE efficiency between PACK and the sender-based TRE is due to their different ability of capturing short-term redundancy in the traffic; PACK cannot capture short-term redundancy while sender-based TRE can.

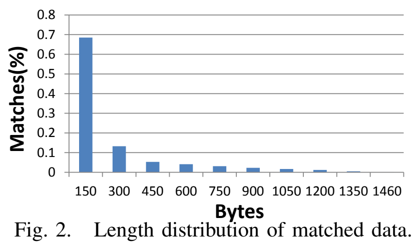
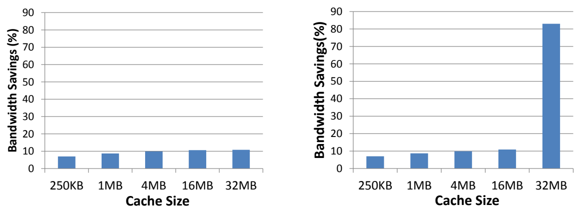


Fig. 2. Length distribution of matched data.



(a) Temporary cache. (b) Persistent cache.

Fig. 3. Detected redundancy in Linux source traffic by sender-based TRE.

Since PACK uses a chunk size of 8KB, it is unable to identify finer-granularity content repetitions and hence misses the short-term redundancy. For example, in Figure 1, the chunk $[CEFA]$ will be sent without any compression due to the false prediction. However, $[CEFA]$ has a great overlap with the previous sent chunk $[ABCD]$. PACK misses such short-term repetition at fine-granularity. By reducing chunk size, PACK may detect fine-granularity redundancy, but it will greatly increase the number of chunks and result in prohibitive prediction transmissions.

2) Long-term redundancy: Eyal et al. [4] have found significant long-term redundancy in YouTube traffic, online social network service traffic and some real-life workloads, where data repetition can occur over 24 hours or months. For the sender-based TRE, a large persistent cache is necessary to capture such long-term redundancy. To verify this point, we investigate the TRE efficiency of two types of sender-based TRE: temporary cache based and persistent cache based, with the traffic generated by downloading 40 Linux source files one by one in their release order, one of real-life workloads used in [4].

In the persistent cache approach, each client is allocated with a cache, which keeps past packets for the entire period of workload downloading in order to detect data repetition across successively downloaded files. In the temporary cache approach, a cache is temporarily allocated with a cache which serves for one file download, which can only capture the redundancy within the file itself. When the download completes, the cache is released and no historical information is stored.

Figure 3 presents the bandwidth savings, i.e., the percentage of total redundancy in the workload traffic detected by the sender-based TRE with various cache sizes. Figure 3(a) shows the redundancy detected by the temporal cache approach with a cache size of 250KB. We see the approach can detect about 7% redundancy inside each file. Increasing cache size yields diminishing returns. It means that most data repetitions occur within the 250KB cache and they actually compose the short-term traffic redundancy that is at the time scale of 250KB/bandwidth. Figure 3(b) shows the redundancy detected by the persistent cache approach with a cache size of 32MB.

Comparing Figure 3(b) with Figure 3(a), we see a significant difference between the temporary cache and persistent cache approaches. As opposed to 10% redundancy detected by temporary cache, persistent cache can detect more than 80% redundancy, in which about 80% redundancy is found in every file except the first downloaded one. We note that the sizes of Linux source files are between 21MB and 31MB. Therefore, the 32MB cache can store a whole file previously downloaded, which enables to detect data repetition across successive files, resulting in more detected redundancy. The redundancy across different versions of Linux source shows a long-term redundancy, because the source file download most likely occurs when a new version is released. Therefore, persistently keeping past transferred data is essential for capturing long-term redundancy. Besides the persistency, the steep rise of detected redundancy in Figure 3(b) also indicates that the cache needs to be large enough to capture a long-term redundancy in the traffic.

However, because the cloud server has limited resources and usually serves a large number of users, it is not practical for the server to maintain a large persistent cache for every client. As a result, the sender-based TRE is unable to efficiently detect long-term redundancy. By offload caching and computation effort from the cloud server to clients, PACK can efficiently capture the long-term redundancy. However, we have shown that it fails to detect the short-term redundancy which is significant in the network traffic in Section III-A1.

Besides, the efficiency of PACK for capturing the long-term redundancy is susceptible to data changes, which can happen frequently in various cloud applications involving frequent data update such as collaborative development [14] and data storage [15]. In PACK, the prediction is true only if the predicted chunk exactly appears at the expected position in the byte stream. Even a small position offset in the sender's outgoing data due to data insertion or deletion can invalidate all the following predictions. For example, in Figure 1, 'E' is inserted into the sender's data. Both the second and third predictions will fail due to this insertion, even though the signature of $[H, I, J, K]$ in sender's data matches the third prediction. Although a hybrid mode of sender-based and receiver-based TRE is proposed for PACK [4] to address disperse data changes, the deployment of two separate TRE schemes causes cumulative costs and implementation complexity. The switching between receiver-based approach and sender-based approach also degrades the capability of either in capturing long-term redundancy or short-term redundancy.

B. Design goals

Based on the above analysis, we conclude that solely using either sender-based or receiver-based solution would fail to eliminate a large amount of redundancy. With the advance of various applications and services in clouds, significant short-term and long-term redundancy can concurrently appear in the network traffic, which calls for a new TRE solution to efficiently capture redundancy at both short-term and long-term time scale. Thus, we aim to design such a TRE scheme, while ensuring the TRE efficiency against data changes and low additional operation cost.

IV. CoRE DESIGN

In this section, we describe the design of CoRE in detail and explain how it achieves redundancy detection in both short-term and long-term time scales.

A. Overview

CoRE has two TRE modules each for capturing short-term redundancy and long-term redundancy respectively. Two TRE modules are integrated to form a two-layer redundancy detection system. For any outbound traffic from the server, CoRE first detects its long-term redundancy by the first-layer TRE module. If no redundancy is found, it turns to the second-layer TRE module to search for short-term redundancy at finer granularity.

The first-layer TRE module detects long-term redundancy by a prediction-based *Chunk-Match* approach like PACK [4]. However, comparing with PACK, we propose an improved prediction algorithm to efficiently handle data changes, so that a small data change will not affect the TRE efficiency. Specifically, the sender in CoRE performs the same chunking algorithm as the receiver, to divide the outgoing data into chunks. Then, for each chunk, the sender looks up its signature in its *prediction store* that keeps all predictions recently received from the receiver. If a matching signature is found, the sender sends a prediction confirmation PRED-ACK message to the receiver. In this way, our prediction matching does not involves data offset, which makes CoRE resilient against data changes.

In the second-layer TRE module, the server maintains a temporary small local cache for each client, and detects short-term redundancy by matching outgoing data with this local cache at fine granularity. In particular, the sender stores its recently transferred chunks in its *chunk cache*. It computes a set of representative fingerprints for each chunk. Every fingerprint, along with a pointer to the corresponding chunk in the chunk cache, is stored in a *fingerprint store*. In this layer, the sender performs *In-Chunk Max-Match* to identify contiguous sub-strings that are repeated across chunks. It checks each representative fingerprint of the chunk against the fingerprint store to find whether the fingerprint already exists there. For each matching fingerprint, the corresponding chunk is retrieved from the chunk cache and the match region is expanded byte-by-byte in both directions to obtain the maximal region of redundant bytes. After that, the sender encodes the matched region in the outgoing chunk with an *in-chunk shim* which contains the signature of the corresponding in-cache chunk, the offset and length of the matched region.

For any incoming packet, the receiver first decodes in-chunk shims if any. If the packet is a PRED-ACK message, the receiver checks the corresponding prediction in its prediction store and finds the expected chunk in its chunk store. Then, it copies the chunk to its TCP input buffer according to the offset in TCP stream specified in PRED-ACK message. An overview of CoRE is given in Figure 4.

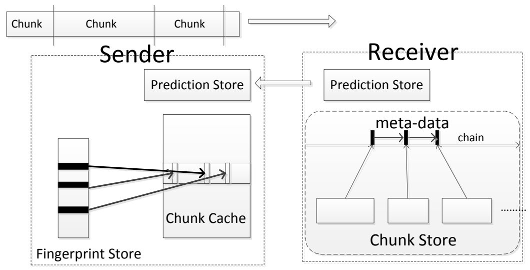


Fig. 4. Overview of CoRE

Below, we explain the details of each of the above components of CoRE.

B. Chunking and Fingerprinting

CoRE performs prediction based Chunk-Match to detect long-term redundancy and local cache based Max-Match to detect short-term redundancy in chunks. In Chunk-Match, the sender divides the data stream into chunks based on data content. It checks the SHA-1 hash value of each outgoing chunk against the prediction store holding the signatures of predicted chunks. Each matched chunk is replaced with a prediction confirmation in the outgoing stream. If an outgoing chunk does not have a matched chunk in the prediction store, the sender conducts Max-Match. In Max-Match, a subset of fingerprints, which are hash values of data in fixed-size windows in the chunk, are computed. The sender compare these fingerprints with its fingerprint store. If a matching fingerprint is found, this means that the data chunk has a window of data that matches an in-cache chunk. The in-cache chunk is retrieved and compared byte-by-byte around the match window to identify the maximal overlap region. Then, the identified region in the outgoing data is replaced with a shim, which indicates its memory offset in cache and its length. After the receiver receives the chunk, it reconstruct the compressed region based on the prediction confirmation and shim.

The prior works [6], [10], [4], [5], [16] use Chunk-Match and Max-Match exclusively. CoRE coordinately uses them to implement two-layer TRE in order to maximally detect redundancy, while achieving prediction efficiency. By using a chunk size at the order of several KB, Chunk-Match identifies and removes redundancy at a large granularity while ensuring the low additional cost at the clients. Only when no matching chunk is found, In-Chunk Max-Match is invoked. By selecting fingerprints with an average interval of 32-64 bytes for each chunk, In-Chunk Max-Match is able to identify redundancy inside chunks at a smaller granularity. In this way, CoRE brought about two benefits. First, successful chunk matches in Chunk-Match eliminate the need for byte-to-byte comparisons in Max-Match, thus saving the computation cost and improving the compression rate at the servers. CoRE has larger chunk size than the previous sender-based TRE, which help to reduce the number of expensive SHA-1 operations and the overhead of chunk-hash storage, and also reduce the number of chunk-hash transmissions between servers and clients. Second, In-Chunk Max-Match can capture the opportunities of redundancy elimination missed by Chunk-Match and improve bandwidth savings.

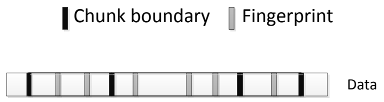


Fig. 5. Chunking and fingerprinting

Current chunking algorithms determine chunk boundaries based on either byte or fingerprint. Byte based algorithms, such as MAXP [6] and Samplebyte [10], choose a byte that satisfies a given condition as chunk boundary. The fingerprint-based algorithms, such as Rabin fingerprint based [10] and PACK chunking [4], compute fingerprints by applying a pseudo-random hash function to sliding windows of w contiguous bytes in a data stream and select a subset of fingerprints with a given sampling frequency. The first byte in the window of each chosen fingerprint forms the boundaries of the chunks. In CoRE, the sender performs both chunking and fingerprinting per chunk. For this purpose, a straight approach is to first use a chunking algorithm to divide a data stream into chunks and then computes the fingerprints within each chunk. However, this approach needs scanning the byte string twice (chunking and fingerprinting), thus increasing the computation cost of the servers. To save the cost, we propose a one single-pass scanning algorithm to generate chunks and fingerprints within chunks, which is based on the chunking algorithm proposed in PACK [4]. By running a fingerprinting algorithm, a set of fingerprints are computed and a subset of them are chosen as chunk boundaries, as shown in Figure 5.

In our algorithm, a XOR-based rolling hash function is used to compute a 64-bit pseudo-random hash value over each sliding window with a size denoted by w. As shown in [4], the XOR-based rolling hash function achieves higher speed than Rabin fingerprinting algorithm for the computation of fingerprints. Therefore, we choose the XOR-based rolling hash function to generate fingerprints and chunk boundaries. Given k specified bit-positions in a 64 bit string, denoted by $P_f = \{b_1, b_2, ..., b_k\}$, the 64-bit pseudo-random hash value with all "1"s at these positions is chosen as a fingerprint. Given a set of n specified bit-positions P_c in a 64-bit string such that $|P_c| = n$ and $P_f \subset P_c$, 64-bit fingerprints with all "1"s at P_c are chosen as chunk boundaries. As a result, the average chunk size is 2^n bytes and the average sampling interval for fingerprints is 2^k bytes. Algorithm 1 shows the pseudo-code of this chunking and fingerprinting algorithm with $w = 48, n = 13, k = 6$.

C. CoRE Sender Algorithm

The sender uses a prediction store to cache predictions from the receiver for a certain time period. Each prediction includes the SHA-1 signature of a predicted chunk and its expected offset, i.e., TCP sequence number in the TCP byte stream. The sender also has a chunk cache to store chunks recently sent. A fingerprint store holds meta-data for representative fingerprints of each cached chunk, which includes the fingerprint value, the position of the chunk referred by the fingerprint, and the

Algorithm 1 Chunking and Fingerprinting

```
1: $cmask \leftarrow 0x00008A3110583080$; //13 1-bits, 8KB chunks
2: $fmask \leftarrow 0x0000000000383080$; //6 1-bits, 64B fingerprint sampling interval
3: $longval \leftarrow 0$; //64-bit
4: for all byte∈ stream do
5:     shift left longval by 1 bit;
6:     $longval \leftarrow longval$ XOR $byte$;
7:     if processed at least 48 bytes and ($longval$ AND $fmask$) == $fmask$ then
8:         found a fingerprint $f$;
9:         if ($longval$ AND $cmask$) == $cmask$ then
10:            $f$ is a chunk boundary;
11:        end if
12:    end if
13: end for
```

byte offset in the chunk where the region represented by the fingerprint starts.

When receiving new data from the upper layer application, the sender performs the chunking and fingerprinting algorithm. For each chunk, if the prediction store is not empty, the sender first computes the SHA-1 signature of the chunk and then looks up it in the prediction store. If a matching signature is found in a prediction p, the sender replaces the outgoing chunk with a PRED-ACK confirmation message which carries a tuple $< offset_p, offset_s >$ where $offset_p$ is the expected offset in prediction p and $offset_s$ is the actual offset of the outgoing chunk in the TCP stream.

In PACK, the sender only stores the current prediction and discards its previously received predictions. Each prediction is compared with current outgoing data over the TCP sequence interval uniquely specified in the prediction. However, the predicted chunks may be very likely to appear in the near future, and the outgoing chunks may not appear at the same position in the chain as predicted at the receiver considering possible disperse insertions, deletions and modifications of chunks. PACK cannot detect the redundancy in these cases. Therefore, we improve the algorithm in PACK by enabling the sender to hold not only recently received predictions but also the previously received predictions. The CoRE sender divides the outgoing data into chunks before matching against predictions in order not to miss redundant chunks in changed data. Each outgoing chunk is compared with all entries in the prediction store regardless of their expected offsets (i.e., TCP sequence).

An outgoing chunk can even match a prediction with an inconsistent TCP sequence. In this way, CoRE can achieve resiliency against data changes and leverage useful predictions as much as possible. For example, in Figure 1, CoRE would divide the outgoing data at the sender into chunks $[ABCD]$ $[BCEFA]$ $[HIJK]$, given the chunking algorithm that determines chunk boundaries based on data content. The signature of chunk $[HIJK]$ is already received as a prediction form the receiver. Then, CoRE finds a match and obtains an opportunity to compress this chunk but PACK misses it.

If the prediction store is empty or the above prediction-based Chunk-Match operation does not find a matching chunk, the sender then performs In-Chunk Max-Match. It checks the fingerprints of the outgoing chunk against the fingerprint store. If a matching fingerprint is found, the corresponding chunk is retrieved from the chunk cache. The data in different windows may generate the same fingerprints. To avoid such possible collision in the fingerprint namespace, the corresponding matched window in the chunk is compared with the outgoing chunk byte-by-byte. Then, the matched window is expanded byte by byte in both directions to obtain the maximal overlapped region. Each matching region in the outgoing chunk is encoded with a shim $<sign_c, offset_c, length, offset_s>$ where $sign_c$ is the signature of the matched chunk in the chunk cache, $offset_c$ is the offset of the matching region in the matching chunk, and $length$ and $offset_s$ are the length and the offset of the matching region in the TCP stream, respectively. If the matching chunk has multiple matching regions, corresponding shims can be compressed together such as $<sign_c, offset_{c1}, length_1, offset_{s1}, offset_{c2}, length_2, offset_{s2}, \ldots>$.

After the sender sends out the chunk, the sender updates the chunk cache and the fingerprint store with this chunk. Figure 6 describes the sender algorithm for processing outgoing data by sate machines. The details of the maintenance of local data structures at the sender are described as follows.

Chunk Cache. The chunk cache is a fixed-size circular FIFO buffer. The server maintains a chunk cache associated with a timer for every client. When the timer expires, the cache is released. Once a new request arrives, a chunk cache is allocated for the corresponding client. The chunk cache stores recently transferred chunks from the server to the client, via either one TCP connection or multiple TCP connections. Once the chunk cache is full, the earliest chunk is evicted and the fingerprints pointing to it are invalidated. Each entry in the chunk cache is a tuple $< data, signature >$, where *data* field is the chunk data and *signature* field is its SHA-1 hash. Note that the signature is not always computed for all chunks. This field is filled for a chunk in two situations. First, when the prediction store is not empty, the outgoing chunk's signature is computed for prediction matching. Second, during the In-Chunk Max-Match, if the *signature* field of a chunk for obtaining the maximal matching region is empty, its signature is computed and filled in the chunk cache. In this way, the expensive signature computation is only performed on-demand, which avoids unnecessary SHA-1 operations and reduces the server's computation cost.

Prediction Store. The server maintains a prediction store for each TCP connection with the client. Once receiving a new prediction from the receiver, the sender inserts it into its prediction store associated with corresponding TCP connection. The prediction store holds the most recent predictions. Outdated predictions need to be identified and evicted to limit the size of the prediction store. For this purpose, in CoRE, we define the elapsed time of a prediction p in the store as:

$$E_p = \begin{cases} 0, & Seq_c \leq_{seq} Seq_p \\ (Seq_c - Seq_p) \bmod 2^{32}, & Seq_c >_{seq} Seq_p \end{cases} \quad (1)$$

where Seq_c is the TCP sequence number of the chunk currently to be sent, and Seq_p is the expected TCP sequence

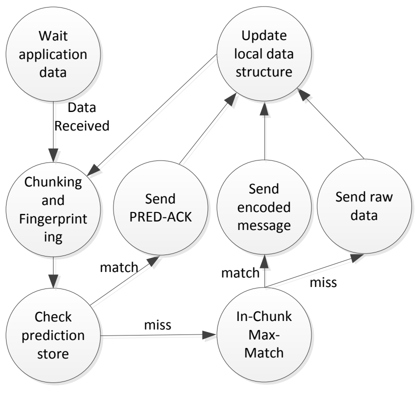


Fig. 6. Sender algorithm

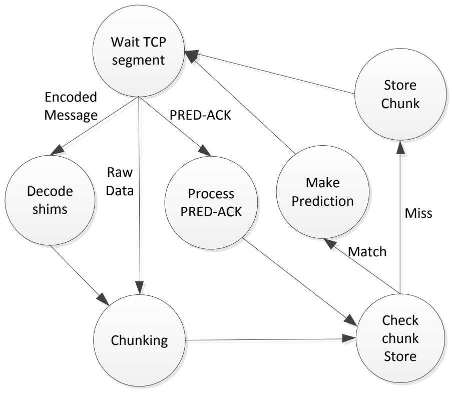


Fig. 7. Receiver algorithm

number in the prediction. $\leq_{seq}$ and $>_{seq}$ are comparisons of TCP sequence number with modulo 2^{32} [17]. We use TTL_{pred} to denote the system parameter for the maximum elapsed time of a prediction. At the time when CoRE compares the outgoing chunk with the TCP sequence number Seq_c against the prediction store, it removes predictions satisfying $E_p > TTL_{pred}$ from the prediction store.

Here, we assume that a larger TCP sequence number difference implies that such prediction has a less relevance with the future outgoing data and vice versa, without the loss of generality. Thus, we use the difference of TCP sequence numbers between a prediction and current outgoing data to measure the timeliness of the prediction. CoRE maintains the predictions within TTL_{pred} rather than only one prediction as in PACK because predictions may appear again in the future due to the repetitive appearance feature of traffic.

D. CoRE Receiver Algorithm

The receiver processes incoming TCP segments according to their different types. There are three types of TCP segments from the sender to the receiver: PRED-ACK message, encoded message with shims and raw data. Upon receiving an PRED-ACK message containing $< offset_p, offset_s >$, the receiver first checks its local prediction store to find the corresponding prediction p which contains the expected offset $offset_p$. Then, it retrieves the chunk expected by prediction p from the chunk store and place the chunk to its TCP input buffer according to the offset $offset_s$ in TCP stream specified by the sender. When receiving the encoded message containing shim <$sign_c$, $offset_c$, *length*, $offset_s$>, the receiver finds the chunk with signature $sign_c$ from its chunk store. The matching region in the chunk indicated by $offset_c$ and *length* is copied to the receiver's TCP input buffer according to the position $offset_s$ in TCP stream.

After decoding the shims or receiving raw data, the receiver performs the chunking operation on the buffered data. The chunking algorithm at the receiver is the same as Algorithm 1 except for fingerprint identification with $fmask$. The chunks from one TCP stream are linked together in the order they are received and cached in the chunk store. If the signature of an incoming chunk is found in the chunk store, the receiver sends to the sender predictions for several subsequent expected chunks. One or multiple predictions are sent in a PRED message. Each prediction consists of the signature of a predicted chunk and its expected offset in TCP stream from the sender to the receiver. Before sending any prediction, the receiver checks whether the TCP sequence range for its predicted chunk has any overlap with anyone in the prediction store. If does, the prediction is discarded. Thus, all sent predictions have no overlap in their TCP sequence range. They are sent in the order of their expected offsets, i.e., TCP sequence numbers. Figure 7 shows the receiver algorithm, in which the receiver enters different processing procedures according to the types of incoming messages.

Prediction Store. The prediction store at the receiver keeps the predictions sent recently. Each entry consists of an expected TCP sequence number in a prediction and a pointer to the corresponding predicted chunk. The chunk signatures in predictions are not stored, since every prediction can be uniquely identified by its expected sequence number. The prediction store is maintained in the similar way as at the sender. When receiving a chunk with TCP sequence number Seq_c either actually received or reconstructed from an incoming PRED-ACK message, the receiver removes any prediction p with $E_p > TTL_{pred}$ from the prediction store before trying to make any new predictions.

V. IMPLEMENTATION

We implemented CoRE in JAVA based on PACK implementation [18]. The protocol is embedded in the TCP options field. The prototype runs on Linux with Netfilter Queue [19].

The implementation of sender component largely follows the discussion in Section IV-C. The implementation of the prediction store at the sender must support quick identification of matching predictions, and efficient identification and deletion of outdated predictions. Considering that the arrivals of predictions are in the increasing order of the expected offset, we use LinkedHashMap because its iteration ordering is normally the order in which keys are inserted. The chunk cache is implemented as a circular FIFO with a maximum of M fixed-size entries. The fingerprint store is implemented as HashMap. When the old chunk is evicted from the chunk cache, the associated fingerprints should be invalidated. To efficiently perform this task, we adopted the method in [12]. We maintain a counter *MaxChunkID* (4 bytes) that increases by one before a new chunk is stored. Each chunk has an unique identifier *ChunkID* which is set to the value of *MaxChunkID* when it is stored. The position of a chunk in the store is computed by *ChunkID%M*, thus we store *ChunkID* instead of the position of the chunk in meta-data of the fingerprint store. For each entry in the fingerprint store, if $ChunkID < MaxChunkID - T$, the corresponding chunk has been evicted and thus the fingerprint is invalid. For the implementation of receiver component, we add the in-chunk shim decoding algorithm, and the prediction store is also implemented as LinkedHashMap.

Parameter Settings. In default, we use an average chunk size of 8KB and fingerprint sampling interval of 64B by setting $n = 13$ and $k = 6$ for the chunking and fingerprinting algorithm. We set $TTL_{pred} = 2^{21}$, $M = 2^9$ for the chunk store at the receiver.

Cache Synchronization. To decode the in-chunk shim, the chunk cache at the sender needs to be synchronized with that at the receiver. Nevertheless, CoRE doesn't require strict synchronization between sender and receiver due to three reasons. First, CoRE operates in TCP stream such that packet loss and disorder can be handled by the TCP protocol. Second, CoRE suppresses the redundancy based on chunks. The drop of a whole 8KB chunk can be easily and early detected by the TCP protocol. Third, as opposed to EndRE [10], the in-chunk shim in CoRE contains the signature of the matching chunk as an index to locate it rather than an address in the cache.

VI. EVALUATION

In this section, we evaluate CoRE and compare it with PACK and the sender-based scheme by both trace-based and testbed approach. Our trace-based evaluation is based on two genuine data set of real-life workloads and two Internet HTTP traffic traces. We deploy CoRE implementation onto our testbed consisting of a server and client(s). Our server is 2 Ghz Dual-Core with 2 GB RAM running Ubuntu, and clients have 2.67GHz Intel Core i5 with 4 GB RAM.

A. Traffic Traces

Our evaluation uses the following trace data sets:

Linux Source Workload (2.0.x): Forty tar files of Linux kernel source code. These files sum up to 1 GB and were released over a period of two year. The traffic is generated by downloading all these tar files in their release order from 2.0.1 to 2.0.40.

IMAP trace: 710MB of IMAP Gmail for past 1 year containing 7500 email messages. These emails consist of personal emails as well as regular mail subscriptions.

Internet traffic traces: We monitored one of the access links from a large university. The link has a 1Gbps full-duplex connection to the backbone and serves roughly 30,000 users. We captured entire packets (including payloads) going in either direction on the link. Due to limited disk volume compared with huge traffic in our measurement architecture, we could not store all kinds of traffic. Thus, without loss of generality, we decided to focus on the web applications as the target traffic by using port 80 to filter them out. However, the HTTP traffic still consume about 1.6GB-2.9GB per minute, which is not suitable for long-term monitoring. In order to study the redundancy in traffic in a long-term period, we adapted our capturing rules for one of the most popular social network (SN) website. As a result, our real Internet traffic traces include one full HTTP trace (1.9GB in 120 seconds) and one SN trace (1.3GB in 2 hours). The detailed information is shown in Table I.

B. TRE efficiency of CoRE

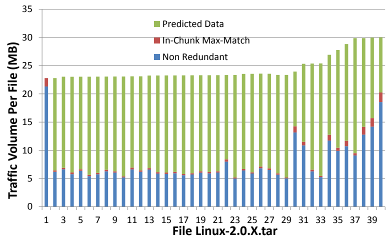
In this section, we first evaluate TRE efficiency of CoRE with the traffics of Linux source files and Gmail messages as in PACK [4], and then compare bandwidth savings of CoRE compared with PACK and sender-based solution. The traffic of Linux source files are resulted from the downloads of these files in the their release order. For the traffic of Gmail, all the email messages are grouped by month, and downloaded by their issue dates. The CoRE sender uses a 4MB chunk cache, a size of the maximum capacity of 512 chunks with an 8KB average chunk size.

Figure 8(a) and 8(b) show the redundancy detected by the prediction-based Chunk-Match and In-Chunk Max-Match of CoRE, respectively. In Figure 8(a), 29 files have more than 70% redundancy and the remaining files have at least 38% redundancy. Total redundancy amounts to 68% in the whole traffic volume of 40 Linux source files, about 664MB. Such a large amount of redundancy is resulted from the high similarity between an earlier version and a subsequent version. When the file "Linux-2.0.1.tar" is downloaded for the first time, there is no similar data that is cached, so that there are no successful predictions and CoRE only found a small redundancy inside the file itself by In-Chunk Max-Match. As we can see, each file contains a little redundancy inside itself and a large amount of redundancy exists across files. This suggests that in such real-world workload, long-term redundancy is significantly more than short-term redundancy. Because a download happens only after the release of a new version and the cache required to detect long-term redundancy has at least the size of a file, the sender-based TRE with small temporary cache cannot obtain bandwidth savings.

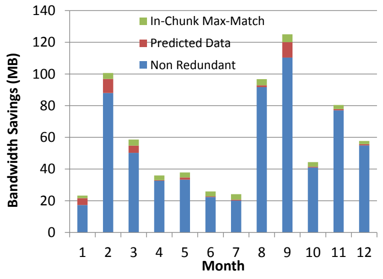
In Figure 8(b), the email traffic in each month has much less redundancy than the Linux source files. The reason is that most of emails are distinct and have much less duplicate contents. The total redundancy detected by CoRE amounts to 11% in our Gmail traffic volume of 12 months, about 71MB, which is less than 31.6% redundancy shown in the Gmail traffic used by PACK [4]. The authors of PACK found that the redundancy arises from large attachments from multiple sources in their Gmail messages. However, our Gmail messages contain few attachments, which causes its less redundancy than PACK's Gmail trace. Little redundancy is detected by prediction in the Gmail traffic of several months from April to August and from October to December, which indicates little long-term redundancy in these Gmail traffics and confirms the distinction of our Gmail messages. We also find that in our detected redundancy, 5% is contributed by prediction and the remaining 6% is contributed by In-Chunk Max-Match. As we can see, the redundancy detected by In-Chunk Max-Match is significant compared to the redundancy detected by prediction,

Trace Name	Description	Dates/ Start Times	Duration	Total Volume(MB)	IP Pairs	Servers(IP)	Clients(IP)
Univ-HTTP	Inbound/outbound http	10am on 11/05/11	120s	1900	8277	3183	1931
Univ-SN	Inbound/outbound for SN	3pm on 11/08/11	2h	1300	3237	6	894

TABLE I
CHARACTERISTICS OF INTERNET TRAFFIC TRACES



(a) Detected redundancy of 40 different Linux Kernel versions



(b) Detected redundancy of 1-year Gmail account by month

Fig. 8. Detected Redundancy by CoRE

Data traffic Name	CoRE	PACK
	% savings	
Linux source	68	54
Email	11	3.6
Univ-HTTP	4.8	0
Univ-SN	8.3	0.3

TABLE II
PERCENTAGE BANDWIDTH SAVINGS OF CORE AND OTHER SOLUTIONS

which is sharp contrast to the results of Linux source files. This indicates that Gmail traffics have significant short-term redundancy, and the receiver-based TRE only obtains less than half of bandwidth savings in our Gmail traffics compared to CoRE. The results verify the necessity of a sender & receiver cooperative TRE.

Table II compares the bandwidth savings for CoRE, PACK and sender-based TRE with different cache sizes. Each entry shows the percentage of bandwidth savings in the total volume of the corresponding traffic trace. From this table, we see that CoRE performs better than PACK because CoRE captures short-term redundancy and is also resilient to data changes.

C. Performance Of CoRE Prediction

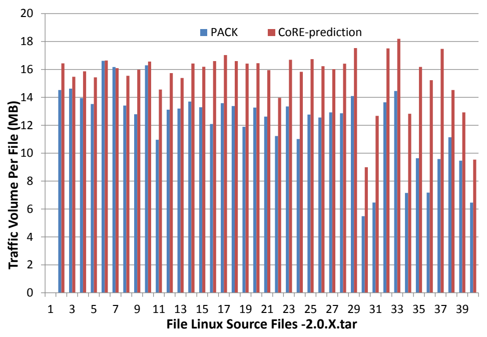
The difference of the prediction designs between CoRE and PACK is that the CoRE sender performs the same chunking operation as the receiver so that it is able to compare chunks with predictions regardless of predicted data offsets in the data stream. As a result, CoRE can ensure TRE efficiency from the changes of outgoing data compared with the original data cached at the receiver. To verify this, we disabled the low layer of In-Chunk Max-Match in CoRE and only measured the redundancy detected by its prediction. We compared the CoRE's prediction solution with PACK's by using Linux source and Gmail traffics, with an average chunk size 8KB for both CoRE and PACK.

Figure 9(a) shows the bandwidth savings, i.e., redundancy in each of the downloaded versions detected by CoRE prediction and PACK prediction, respectively. Figure 9(b) shows the redundancy in each month of the email messages. The experimental results show that CoRE prediction scheme can detect and eliminate more redundancy than PACK. The amount of redundancy detected by CoRE prediction is 21% higher than that detected by PACK in the traffic of Linux source files, and 22% higher in Gmail traffic. The reason for the higher performance of CoRE prediction is that it is common that the update of the Linux source code involves the new code insertion and old code deletion. In the Gmail traffic, the emails usually contain short-term data repetition because when people reply to an email, the content of this email is usually included to the outgoing message.

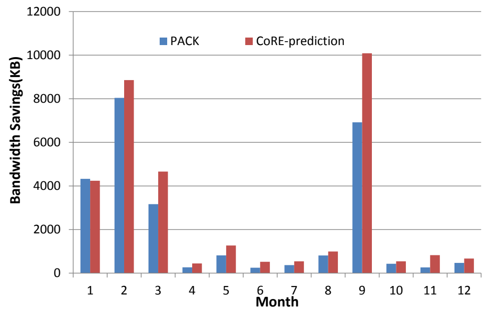
Resiliency against data changes To further verify the resiliency of CoRE prediction against data changes, we chose a 31.5MB Linux tar file and inserted one random byte to the random positions into the file. By changing the average distance between two successive inserting positions in the file, we changed the degree of data changes on the original file. We downloaded the original file first, and then downloaded the modified file to measure its redundancy that CoRE and PACK can detect given the cache of the original file at the receiver. Figure 10 shows that CoRE prediction detects more redundancy than PACK under various degrees of data changes. "No changes" in the x-axis means that the second file we downloaded is the same as the original file. In this case, CoRE and PACK detect the same amount of redundancy. As we can see, PACK is very susceptible to data changes. It decreases exponentially as the insertion intensity increases. In contrast, CoRE is much more resilient against data changes. It decreases linearly as the insertion intensity increases. The results confirm the resiliency of CoRE prediction to data changes due to the data chunking at the sender and the matching with the historical predictions.

D. Cost Evaluation

To measure the operation cost of CoRE at the sender and receiver, we monitored CPU utilization on the server and the client at an interval of one second. Since higher redundancy



(a) Bandwidth savings of 40 different Linux Kernel versions



(b) Bandwidth savings of 1-year Gmail account by month

Fig. 9. Comparison of CoRE prediction and PACK

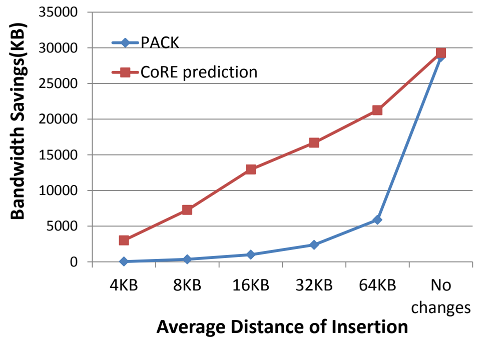


Fig. 10. Detected redundancy VS Insertion intensity

could cause more prediction and elimination cost, we use the Linux source workload to measure the TRE operation cost. Table III compares CoRE's server CPU utilization ratio with PACK and sender-based approach As we can see, CoRE has a little higher CPU utilization ratio than PACK, and both of them are less than sender-based solution. Such results indicate that chunking and fingerprinting in CoRE incur a low additional cost compared to PACK. Therefore, CoRE can still efficiently obtain overall gain savings when used in cloud environment.

VII. CONCLUSION

By the real trace driven study on existing end-to-end sender-side and receiver-side TRE solutions, we identify their limitations for capturing redundancy in short-term and long-term data redundancy. Thus, we propose an Cooperative end-to-end TRE CoRE, which integrates both sender-side and receiver-side efforts. Through extensive trace-driven experiments, we show that CoRE is able to capture both short-term and long-term redundancy, and can eliminate much more redundancy than PACK while incurring a low additional operation cost.

Scheme Name	Server CPU Utilization %	Client CPU Utilization %
CoRE	3.41	2.75
PACK	2.92	2.29
Sender-based	5.28	-

TABLE III
CPU COST

ACKNOWLEDGEMENTS

This research was supported in part by U.S. NSF grants CNS-1249603, OCI-1064230, CNS-1049947, CNS-1156875, CNS-0917056 and CNS-1057530, CNS-1025652, CNS-0938189, CSR-2008826, CSR-2008827, Microsoft Research Faculty Fellowship 8300751, and U.S. Department of Energy's Oak Ridge National Laboratory including the Extreme Scale Systems Center located at ORNL and DoD 4000111689.

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, pp. 50–58, April 2010.
[2] "The hidden cost of the cloud: Bandwidth charges," http://gigaom.com/2009/07/17/the-hidden-cost-of-the-cloud-bandwidth-charges/.
[3] "Cloud bandwidth costs & the true cost of cloud hosting," http://www.earthlinkcloud.com/2011/09/cloud-bandwidth-costs-the-true-cost-of-cloud-hosting/.
[4] E. Zohar, I. Cidon, and O. O. Mokryn, "The power of prediction: cloud bandwidth and cost reduction," in *ACM SIGCOMM*, 2011, pp. 86–97.
[5] N. T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," in *ACM SIGCOMM*, 2000, pp. 87–95.
[6] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee, "Redundancy in network traffic: findings and implications," in *SIGMETRICS/Performance*, 2009, pp. 37–48.
[7] "Riverbed networks : Wan optimization." http://www.riverbed.com/solutions/optimize.
[8] "Juniper networks: Application acceleration." http://www.juniper.net/us/en/products-services/application-acceleration/, 1996.
[9] "Cisco wide are application acceleration services," http://www.cisco.com/en/US/products/ps5680/ Products_Sub_Category_Home.html.
[10] B. Agarwal, 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, pp. 419–432.
[11] M.Rabin, "Fingerprinting by random polynomials," *Technical report Harvard University*, vol. TR-15-81, 1981.
[12] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker, "Packet caches on routers: the implications of universal redundant traffic elimination," in *ACM SIGCOMM*. ACM, 2008, pp. 219–230.
[13] A. Anand, V. Sekar, and A. Akella, "Smartre: an architecture for coordinated network-wide redundancy elimination," in *ACM SIGCOMM*. ACM, 2009, pp. 87–98.
[14] "Paas." [Online]. Available: http://www.ibm.com/cloud-computing/us/en/paas.html
[15] "Dropbox." [Online]. Available: www.dropbox.com
[16] S. Ihm, K. Park, and V. S. Pai, "Wide-area network acceleration for the developing world," in *USENIX annual technical conference*. USENIX Association, 2010, pp. 18–18.
[17] G. Wright and W. Stevens., *TCP/IP Illustrated, Volume2: The Implementation*. Addison-Wesley,Massachusetts, 1995.
[18] "Pack source code," http://www.venus-c.eu/pages/partner.aspx?id=10.
[19] "netfilter/iptables:libnetfilter_queue," http://www.netfilter.org/projects/libnetfilter_queue, Oct 2005.