The Power of Prediction: Cloud Bandwidth and Cost Reduction

Eyal Zohar*
Technion - Israel Institute of Technology
eyalzo@tx.technion.ac.il

Israel Cidon
Technion - Israel Institute of Technology
cidon@ee.technion.ac.il

Osnat (Ossi) Mokryn†
Tel Aviv Academic College
ossi@mta.ac.il

ABSTRACT

In this paper we present PACK (Predictive ACKs), a novel end-to-end Traffic Redundancy Elimination (TRE) system, designed for cloud computing customers.

Cloud-based TRE needs to apply a judicious use of cloud resources so that the bandwidth cost reduction combined with the additional cost of TRE computation and storage would be optimized. PACK's main advantage is its capability of offloading the cloud-server TRE effort to end-clients, thus minimizing the processing costs induced by the TRE algorithm.

Unlike previous solutions, PACK does not require the server to continuously maintain clients' status. This makes PACK very suitable for pervasive computation environments that combine client mobility and server migration to maintain cloud elasticity.

PACK is based on a novel TRE technique, which allows the client to use newly received chunks to identify previously received chunk chains, which in turn can be used as reliable predictors to future transmitted chunks.

We present a fully functional PACK implementation, transparent to all TCP-based applications and network devices. Finally, we analyze PACK benefits for cloud users, using traffic traces from various sources.

Categories and Subject Descriptors

C.2.m [**Computer-Communication Networks**]: Miscellaneous

General Terms

Algorithms, Design, Measurement

Keywords

Caching, Cloud computing, Network optimization, Traffic redundancy elimination

*Also with HPI Research School.
†Also with Technion - Israel Institute of Technology.

SIGCOMM'11, August 15–19, 2011, Toronto, Ontario, Canada.

1. INTRODUCTION

Cloud computing offers its customers an economical and convenient *pay as you go* service model, known also as *usage-based pricing* [6]. Cloud customers[1] pay only for the actual use of computing resources, storage and bandwidth, according to their changing needs, utilizing the cloud's scalable and elastic computational capabilities. Consequently, cloud customers, applying a judicious use of the cloud's resources, are motivated to use various traffic reduction techniques, in particular Traffic Redundancy Elimination (TRE), for reducing bandwidth costs.

Traffic redundancy stems from common end-users' activities, such as repeatedly accessing, downloading, distributing and modifying the same or similar information items (documents, data, web and video). TRE is used to eliminate the transmission of redundant content and, therefore, to significantly reduce the network cost. In most common TRE solutions, both the sender and the receiver examine and compare signatures of data chunks, parsed according to the data content prior to their transmission. When redundant chunks are detected, the sender replaces the transmission of each redundant chunk with its strong signature [16, 23, 20]. Commercial TRE solutions are popular at enterprise networks, and involve the deployment of two or more proprietary protocol, state synchronized middle-boxes at both the intranet entry points of data centers and branch offices, eliminating repetitive traffic between them (e.g., Cisco [15], Riverbed [18], Quantum [24], Juniper [14], Bluecoat [7], Expand Networks [9] and F5 [10]).

While proprietary middle-boxes are popular point solutions within enterprises, they are not as attractive in a cloud environment. First, cloud providers cannot benefit from a technology whose goal is to reduce customer bandwidth bills, and thus are not likely to invest in one. Moreover, a fixed client-side and server-side middle-box pair solution is inefficient for a combination of a mobile environment, which detaches the client from a fixed location, and cloud-side elasticity which motivates work distribution and migration among data centers. Therefore, it is commonly agreed that a universal, software-based, end-to-end TRE is crucial in today's pervasive environment [4, 1]. This enables the use of a standard protocol stack and makes a TRE within end-to-end secured traffic (e.g., SSL) possible.

In this paper, we show that cloud elasticity calls for a new TRE solution that does not require the server to continuously maintain clients' status. First, cloud load balancing and power optimizations may lead to a server-side process and data migration environment, in which TRE solutions that require full synchronization between the server and the client are hard to accomplish or may lose ef-

[1]We refer as *cloud customers* to organizations that export services to the cloud, and as *users* to the end-users and devices that consume the service

ficiency due to lost synchronization. Moreover, the popularity of rich media that consume high bandwidth motivates CDN solutions, in which the service point for fixed and mobile users may change dynamically according to the relative service point locations and loads.

Finally, if an end-to-end solution is employed, its additional computational and storage costs at the cloud-side should be weighed against its bandwidth saving gains. Clearly, a TRE solution that puts most of its computational effort on the cloud-side[2] may turn to be less cost-effective than the one that leverages the combined client-side capabilities. Given an end-to-end solution, we have found through our experiments that sender-based end-to-end TRE solutions [23, 1] add a considerable load to the servers, which may eradicate the cloud cost saving addressed by the TRE in the first place. Moreover, our experiments further show that current end-to-end solutions also suffer from the requirement to maintain end-to-end synchronization that may result in degraded TRE efficiency.

In this paper, we present a novel receiver-based end-to-end TRE solution that relies on the power of predictions to eliminate redundant traffic between the cloud and its end-users. In this solution, each receiver observes the incoming stream and tries to match its chunks with a previously received chunk chain or a chunk chain of a local file. Using the long-term chunks meta-data information kept locally, the receiver sends to the server predictions that include chunks' signatures and easy to verify hints of the sender's future data. Upon a hint match the sender triggers the TRE operation, saving the cloud's TRE computational effort in the absence of traffic redundancy.

Offloading the computational effort from the cloud to a large group of clients forms a load distribution action, as each client processes only its TRE part. The receiver-based TRE solution addresses mobility problems common to quasi-mobile desktop/laptops computational environments. One of them is cloud elasticity due to which the servers are dynamically relocated around the federated cloud, thus causing clients to interact with multiple changing servers. Another property is IP dynamics, which compel roaming users to frequently change IP addresses. In addition to the receiver-based operation, we also suggest a hybrid approach, which allows a battery powered mobile device to shift the TRE computation overhead back to the cloud by triggering a sender-based end-to-end TRE similar to [1].

To validate the receiver-based TRE concept, we implemented, tested and performed realistic experiments with PACK within a cloud environment. Our experiments demonstrate a cloud cost reduction achieved at a reasonable client effort while gaining additional bandwidth savings at the client side. The implementation code, over 25,000 lines of C and Java, can be obtained from [22]. Our implementation utilizes the TCP Options field, supporting all TCP-based applications such as web, video streaming, P2P, email, etc.

We propose a new computationally light-weight chunking (fingerprinting) scheme termed *PACK chunking*. PACK chunking is a new alternative for Rabin fingerprinting traditionally used by RE applications. Experiments show that our approach can reach data processing speeds over 3 Gbps, at least 20% faster than Rabin fingerprinting.

In addition, we evaluate our solution and compare it to previous end-to-end solutions using tera-bytes of real video traffic consumed by 40,000 distinct clients, captured within an ISP, and traffic obtained in a social network service for over a month. We demonstrate that our solution achieves 30% redundancy elimination without significantly affecting the computational effort of the sender, resulting in a 20% reduction of the overall cost to the cloud customer.

The paper is organized as follows: Section 2 reviews existing TRE solutions. In Section 3 we present our receiver-based TRE solution and explain the prediction process and the prediction-based TRE mechanism. In Section 4 we present optimizations to the receiver-side algorithms. Section 5 evaluates data redundancy in a cloud and compares PACK to sender-based TRE. Section 6 details our implementation and discusses our experiments and results.

2. RELATED WORK

Several TRE techniques have been explored in recent years. A protocol-independent TRE was proposed in [23]. The paper describes a packet-level TRE, utilizing the algorithms presented in [16].

Several commercial TRE solutions described in [15] and [18], have combined the sender-based TRE ideas of [23] with the algorithmic and implementation approach of [20] along with protocol specific optimizations for middle-boxes solutions. In particular, [15] describes how to get away with three-way handshake between the sender and the receiver if a full state synchronization is maintained.

[3] and [5] present redundancy-aware routing algorithm. These papers assume that the routers are equipped with data caches, and that they search those routes that make a better use of the cached data.

A large-scale study of real-life traffic redundancy is presented in [11], [25] and [4]. Our paper builds on the latter's finding that "an end to end redundancy elimination solution, could obtain most of the middle-box's bandwidth savings", motivating the benefit of low cost software end-to-end solutions.

Wanax [12] is a TRE system for the developing world where storage and WAN bandwidth are scarce. It is a software-based middle-box replacement for the expensive commercial hardware. In this scheme, the sender middle-box holds back the TCP stream and sends data signatures to the receiver middle-box. The receiver checks whether the data is found in its local cache. Data chunks that are not found in the cache are fetched from the sender middle-box or a nearby receiver middle-box. Naturally, such a scheme incurs a three-way-handshake latency for non-cached data.

EndRE [1] is a sender-based end-to-end TRE for enterprise networks. It uses a new chunking scheme that is faster than the commonly-used Rabin fingerprint, but is restricted to chunks as small as 32-64 bytes. Unlike PACK, EndRE requires the server to maintain a fully and reliably synchronized cache for each client. To adhere with the server's memory requirements these caches are kept small (around 10 MB per client), making the system inadequate for medium-to-large content or long-term redundancy. EndRE is server specific, hence not suitable for a CDN or cloud environment.

To the best of our knowledge none of the previous works have addressed the requirements for a cloud computing friendly, end-to-end TRE which forms PACK's focus.

3. THE PACK ALGORITHM

For the sake of clarity, we first describe the basic receiver-driven operation of the PACK protocol. Several enhancements and optimizations are introduced in Section 4.

The stream of data received at the PACK receiver is parsed to a sequence of variable size, content-based signed chunks similar to [16][20]. The chunks are then compared to the receiver local storage, termed *chunk store*. If a matching chunk is found in the local chunk store, the receiver retrieves the sequence of subsequent chunks, referred to as a *chain*, by traversing the sequence of LRU

[2]We assume throughout the paper that the cloud-side, following the current Web service model, is dominated by a sender operation.

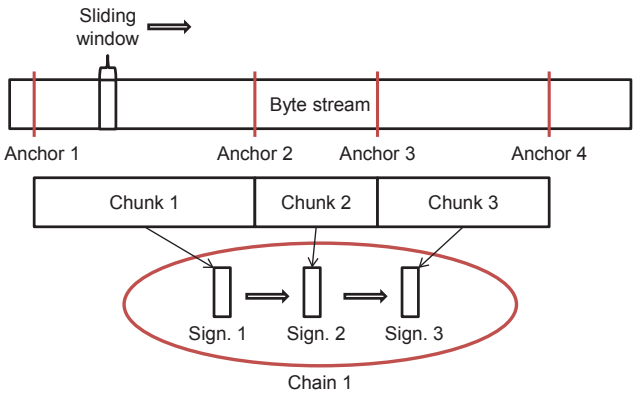


Figure 1: From stream to chain

chunk pointers that are included in the chunks' metadata. Using the constructed chain, the receiver sends a prediction to the sender for the subsequent data. Part of each chunk's prediction, termed a *hint*, is an easy to compute function with a small enough false-positive value, such as the value of the last byte in the predicted data or a byte-wide XOR checksum of all or selected bytes. The prediction sent to the receiver includes the range of the predicted data, the hint and the signature of the chunk. The sender identifies the predicted range in its buffered data, and verifies the hint for that range. If the result matches the received hint, it continues to perform the more computationally intensive SHA-1 signature operation. Upon a signature match, the sender sends a confirmation message to the receiver, enabling it to copy the matched data from its local storage.

3.1 Receiver Chunk Store

PACK uses a new *chains* scheme, described in Figure 1, in which chunks are linked to other chunks according to their last received order. The PACK receiver maintains a *chunk store*, which is a large size cache of chunks and their associated meta-data. Chunk's meta-data includes the chunk's signature and a (single) pointer to the successive chunk in the last received stream containing this chunk. Caching and indexing techniques are employed to efficiently maintain and retrieve the stored chunks, their signatures and the chains formed by traversing the chunk pointers.

When the new data is received and parsed to chunks, the receiver computes each chunk's signature using SHA-1. At this point, the chunk and its signature are added to the chunk store. In addition, the meta-data of the previously received chunk in the same stream is updated to point to the current chunk.

The unsynchronized nature of PACK allows the receiver to map each existing file in the local file system to a chain of chunks, saving in the chunk store only the meta-data associated with the chunks. [3] Using the latter observation, the receiver can also share chunks with peer clients within the same local network utilizing a simple map of network drives.

3.1.1 Chunk Size

The utilization of a small chunk size presents better redundancy elimination when data modifications are fine-grained, such as sporadic changes in a HTML page. On the other hand, the use of smaller chunks increases the storage index size, memory usage and magnetic disk seeks. It also increases the transmission overhead of the virtual data exchanged between the client and the server.

[3]De-duplicated storage systems provide similar functionality and can be used for this purpose.

Unlike IP level TRE solutions that are limited by the IP packet size (∼1,500 bytes), PACK operates on TCP streams, and can, therefore, handle large chunks and entire chains. Although our design permits each PACK client to use any chunk size, we recommend an average chunk size of 8KB (see Section 6).

3.2 The Receiver Algorithm

Upon the arrival of new data, the receiver computes the respective signature for each chunk, and looks for a match in its local chunk store. If the chunk's signature is found, the receiver determines whether it is a part of a formerly received chain, using the chunks' meta-data. If affirmative, the receiver sends a prediction to the sender for several next expected chain chunks. The prediction carries a starting point in the byte stream (i.e., offset) and the identity of several subsequent chunks (PRED command).

Upon a successful prediction, the sender responds with a PRED-ACK confirmation message. Once the PRED-ACK message is received and processed, the receiver copies the corresponding data from the chunk store to its TCP input buffers, placing it according to the corresponding sequence numbers. At this point, the receiver sends a normal TCP ACK with the next expected TCP sequence number. In case the prediction is false, or one or more predicted chunks are already sent, the sender continues with normal operation, e.g., sending the raw data, without sending a PRED-ACK message.

Proc. 1 Receiver Segment Processing

```
1. if segment carries payload data then
2.     calculate chunk
3.     if reached chunk boundary then
4.         activate predAttempt()
5.     end if
6. else if PRED-ACK segment then
7.     processPredAck()
8.     activate predAttempt()
9. end if
```

Proc. 2 predAttempt()

```
1. if received chunk matches one in chunk store then
2.     if foundChain(chunk) then
3.         prepare PREDs
4.         send single TCP ACK with PREDs according to Options
           free space
5.         exit
6.     end if
7. else
8.     store chunk
9.     link chunk to current chain
10. end if
11. send TCP ACK only
```

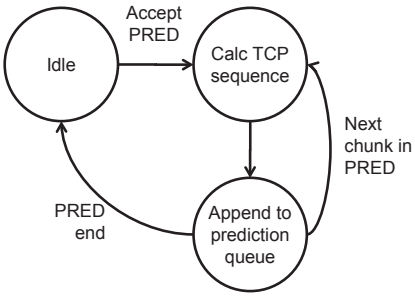
Proc. 3 processPredAck()

```
1. for all offset ∈ PRED-ACK do
2.     read data from chunk store
3.     put data in TCP input buffer
4. end for
```

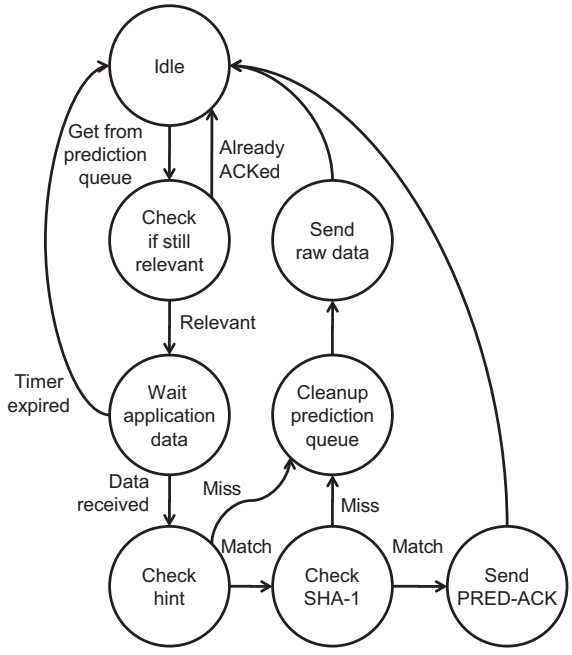
3.3 The Sender Algorithm

When a sender receives a PRED message from the receiver, it tries to match the received predictions to its buffered (yet to be sent) data. For each prediction, the sender determines the corresponding TCP sequence range and verifies the hint. Upon a hint match, the sender calculates the more computationally intensive SHA-1 signature for the predicted data range, and compares the result to the signature received in the PRED message. Note that in case the hint does not match, a computationally expansive operation is saved. If the two SHA-1 signatures match, the sender can safely assume that the receiver's prediction is correct. In this case, it replaces the corresponding outgoing buffered data with a PRED-ACK message.

Figure 2 illustrates the sender operation using state machines. Figure 2a describes the parsing of a received PRED command. Figure 2b describes how the sender attempts to match a predicted range to its outgoing data. First, it finds out if this range has been already sent or not. In case the range has already been acknowledged, the corresponding prediction is discarded. Otherwise, it tries to match the prediction to the data in its outgoing TCP buffers.



(a) Filling the prediction queue



(b) Processing the prediction queue and sending PRED-ACK or raw data

Figure 2: Sender algorithms

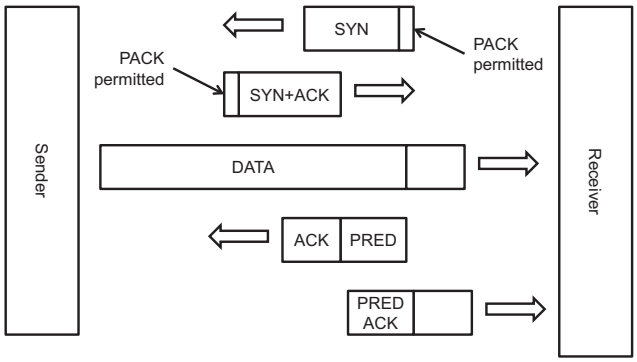


Figure 3: PACK wired protocol in a nutshell

3.4 The Wired Protocol

In order to conform with existing firewalls and minimize overheads, we use the TCP Options field to carry the PACK wired protocol. It is clear that PACK can also be implemented above the TCP level while using similar message types and control fields.

Figure 3 illustrates the way the PACK wired protocol operates under the assumption that the data is redundant. First, both sides enable the PACK option during the initial TCP handshake by adding a *PACK permitted* flag (denoted by a bold line) to the TCP Options field. Then, the sender sends the (redundant) data in one or more TCP segments, and the receiver identifies that a currently received chunk is identical to a chunk in its chunk store. The receiver, in turn, triggers a TCP ACK message and includes the prediction in the packet's Options field. Last, the sender sends a confirmation message (PRED-ACK) replacing the actual data.

4. OPTIMIZATIONS

For the sake of clarity, the previous section presents the most basic version of the PACK protocol. In the following one, we describe additional options and optimizations.

4.1 Adaptive Receiver Virtual Window

PACK enables the receiver to locally obtain the sender's data when a local copy is available, thus eliminating the need to send this data through the network. We term the receiver's fetching of such local data as the reception of *virtual data*.

When the sender transmits a high volume of virtual data, the connection rate may be, to a certain extent, limited by the number of predictions sent by the receiver. This, in turn, means that the receiver predictions and the sender confirmations should be expedited in order to reach high virtual data rate. For example, in case of a repetitive success in predictions, the receiver's side algorithm may become optimistic and gradually increase the ranges of its predictions, similarly to the TCP rate adjustment procedures.

PACK enables a large prediction size by either sending several successive PRED commands or by enlarging PRED command range to cover several chunks.

PACK enables the receiver to combine several chunks into a single range, as the sender is not bounded to the anchors originally used by the receiver's data chunking algorithm. The combined range has a new hint and a new signature that is a SHA-1 of the concatenated content of the chunks.

The variable prediction size introduces the notion of a *virtual window*, which is the current receiver's window for virtual data. The virtual window is the receiver's upper bound for the aggregated number of bytes in all the pending predictions. The virtual

Proc. 4 predAttemptAdaptive() - obsoletes Proc. 2

```
1. {new code for Adaptive}
2. if received chunk overlaps recently sent prediction then
3.    if received chunk matches the prediction then
4.       predSizeExponent()
5.    else
6.       predSizeReset()
7.    end if
8. end if
9. if received chunk matches one in signature cache then
10.   if foundChain(chunk) then
11.      {new code for Adaptive}
12.      prepare PREDs according to predSize
13.      send TCP ACKs with all PREDs
14.      exit
15.   end if
16. else
17.   store chunk
18.   append chunk to current chain
19. end if
20. send TCP ACK only
```

Proc. 5 processPredAckAdaptive() - obsoletes Proc. 3

```
1. for all offset ∈ PRED-ACK do
2.    read data from disk
3.    put data in TCP input buffer
4. end for
5. {new code for Adaptive}
6. predSizeExponent()
```

window is first set to a minimal value, which is identical to the receiver's flow control window. The receiver increases the virtual window with each prediction success, according to the following description.

Upon the first chunk match, the receiver sends predictions limited to its initial virtual window. It is likely that, before the predictions arrive at the sender, some of the corresponding real data is already transmitted from it. When the real data arrives, the receiver can partially confirm its prediction and increase the virtual window. Upon getting PRED-ACK confirmations from the sender, the receiver also increases the virtual window. This logic resembles the slow-start part of the TCP rate control algorithm. When a mismatch occurs, the receiver switches back to the initial virtual window.

Proc. 4 describes the advanced algorithm performed at the receiver's side. The code at lines 2-8 describes PACK behavior when a data segment arrives after its prediction was sent and the virtual window is doubled. Proc. 5 describes the reception of a successful acknowledgement message (PRED-ACK) from the sender. The receiver reads the data from the local chunk store. It then modifies the next byte sequence number to the last byte of the redundant data that has just been read plus one, and sends the next TCP ACK, piggybacked with the new prediction. Finally, the virtual window is doubled.

The size increase of the virtual window introduces a trade-off in case the prediction fails from some point on. The code in Proc. 4 line 6 describes the receiver's behavior when the arriving data does not match the recently sent predictions. The new received chunk may, of course, start a new chain match. Following the reception of the data, the receiver reverts to the initial virtual window (conforming to the normal TCP receiver window size) until a new match

Proc. 6 Receiver Segment Processing Hybrid - obsoletes Proc. 1

```
1. if segment carries payload data then
2.    calculate chunk
3.    if reached chunk boundary then
4.       activate predAttempt()
5.       {new code for Hybrid}
6.       if detected broken chain then
7.          calcDispersion(255)
8.       else
9.          calcDispersion(0)
10.      end if
11.   end if
12. else if PRED-ACK segment then
13.   processPredAck()
14.   activate predAttempt()
15. end if
```

Proc. 7 processPredAckHybrid() - obsoletes Proc. 3

```
1. for all offset ∈ PRED-ACK do
2.    read data from disk
3.    put data in TCP input buffer
4.    {new code for Hybrid}
5.    for all chunk ∈ offset do
6.       calcDispersion(0)
7.    end for
8. end for
```

is found in the chunk store. Note that even a slight change in the sender's data, compared with the saved chain, causes the entire prediction range to be sent to the receiver as raw data. Hence, using large virtual windows introduces a tradeoff between the potential rate gain and the recovery effort in the case of a missed prediction.

4.2 The Hybrid Approach

PACK's receiver-based mode is less efficient if changes in the data are scattered. In this case, the prediction sequences are frequently interrupted, which, in turn, forces the sender to revert to raw data transmission until a new match is found at the receiver and reported back to the sender. To that end, we present the PACK hybrid mode of operation. When PACK recognizes a pattern of dispersed changes, it may select to trigger a sender-driven approach in the spirit of [23][15][18][5].

However, as was explained earlier, we would like to revert to the sender-driven mode with a minimal computational and buffering overhead at the server in the steady state. Therefore, our approach is to first evaluate at the receiver the need for a sender-driven operation and then to report it back to the sender. At this point, the sender can decide if it has enough resources to process a sender-driven TRE for some of its clients. To support this enhancement, an additional command (DISPER) is introduced. Using this command, the receiver periodically sends its estimated level of dispersion, ranging from 0 for long smooth chains, up to 255.

PACK computes the data dispersion value using an exponential smoothing function:

$$D \leftarrow \alpha D + (1 - \alpha)M \tag{1}$$

Where α is a smoothing factor. The value M is set to 0 when a chain break is detected and 255 otherwise.

5. EVALUATION

The objective of this section is twofold: evaluating the potential data redundancy for several applications that are likely to reside in a cloud, and to estimate the PACK performance and cloud costs of the redundancy elimination process.

Our evaluations are conducted using (i) video traces captured at a major ISP, (ii) traffic obtained from a popular social network service, and (iii) genuine data sets of real-life workloads. In this section, we relate to an average chunk size of 8 KB, although our algorithm allows each client to use a different chunk size.

5.1 Traffic Redundancy

5.1.1 Traffic Traces

We obtained a 24 hours recording of traffic at an ISP's 10 Gbps PoP router, using a 2.4 GHz CPU recording machine with 2 TB storage (4 x 500 GB 7,200 RPM disks) and 1 Gbps NIC. We filtered YouTube traffic using deep packet inspection, and mirrored traffic associated with YouTube servers IP addresses to our recording device. Our measurements show that YouTube traffic accounts for 13% of the total daily web traffic volume of this ISP. The recording of the full YouTube stream would require 3 times our network and disk write speeds. Therefore, we isolated 1/6 of the obtained YouTube traffic, grouped by the video identifier (keeping the redundancy level intact) using a programmed load balancer that examined the upstream HTTP requests and redirected downstream sessions according to the video identifier that was found in the YouTube's URLs, to a total of 1.55 TB. We further filtered out the client IP addresses that were used too intensively to represent a single user, and were assumed to represent a NAT address.

Note that YouTube's video content is not cacheable by standard Web proxies since its URL contains private single-use tokens changed with each HTTP request. Moreover, most Web browsers cannot cache and reuse partial movie downloads that occur when end-users skip within a movie, or switch to another movie before the previous one ends.

Table 1 summarizes our findings. We recorded more than 146K distinct sessions, in which 37K users request over 39K distinct movies. Average movie size is 15 MB while the average session size is 12 MB, with the difference stemming from end-user skips and interrupts. When the data is sliced into 8 KB chunks, PACK brings a traffic savings of up to 30%, assuming the end-users start with an empty cache, which is a worst case scenario.

Figure 4 presents the YouTube traffic and the redundancy obtained by PACK over the entire period, with the redundancy sampled every 10 minutes and averaged. This end-to-end redundancy arises solely from self similarity in the traffic created by end-users. We further analyzed these cases and found that end-users very often download the same movie or parts of it repeatedly. The latter is mainly an inter-session redundancy produced by end-users that skip forward and backward in a movie and producing several (partially) overlapping downloads. Such skips occurred at 15% of the sessions and mostly in long movies (over 50 MB).

Since we assume the cache is empty at the beginning, it takes a while for the chunk cache to fill up and enter a steady state. In the steady state, around 30% of the traffic is identified as redundant and removed. We explain the length of the warm-up time by the fact that YouTube allows browsers to cache movies for 4 hours, which results in some replays that do not produce downloads at all.

Table 1: Data and PACK's results of 24 hours YouTube traffic trace

	Recorded
Traffic volume	1.55TB
Max speed	473Mbps
Est. PACK TRE	29.55%
Sessions	146,434
Unique videos	39,478
Client IPs	37,081

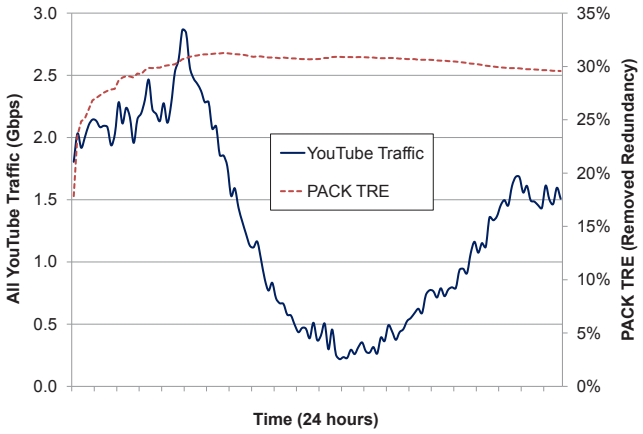


Figure 4: ISP's YouTube traffic over 24 hours, and PACK redundancy elimination ratio with this data

5.1.2 Static Data-set

We acquired the following static data-sets:

Linux source Different Linux kernel versions: all the forty 2.0.x tar files of the kernel source code that sum up to 1 GB.

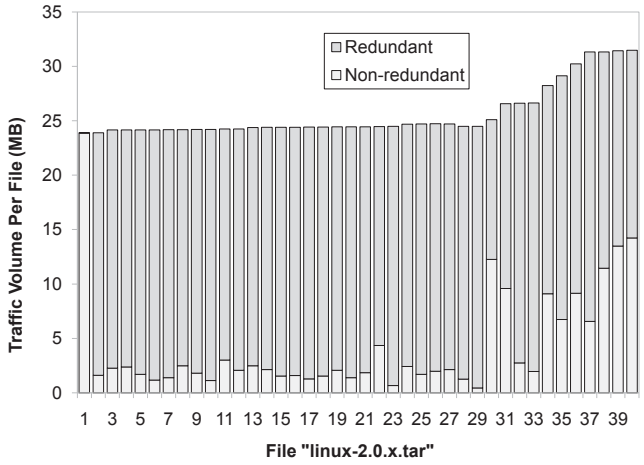
Email A single-user Gmail account with 1,140 email messages over a year, that sum up to 1.09 GB.

The 40 Linux source versions were released over a period of two years. All tar files in the original release order, from 2.0.1 to 2.0.40, were downloaded to a download directory, mapped by PACK, to measure the amount of redundancy in the resulted traffic. Figure 5a shows the redundancy in each of the downloaded versions. Altogether the Linux source files show 83.1% redundancy, which accounts to 830 MB.

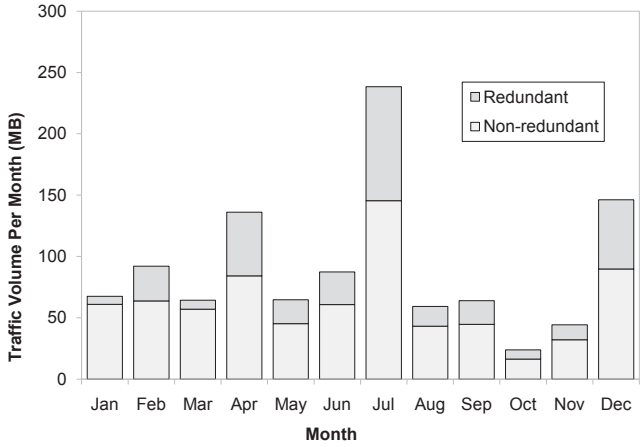
To obtain an estimate of the redundancy in email traffic we operated an IMAP client that fully synchronized the remote Gmail account with a new local folder. Figure 5b shows the redundancy in each month, according to the email message's issue date. The total measured traffic redundancy was 31.6%, which is roughly 350 MB. We found this redundancy to arise from large attachments that are sent by multiple sources, email correspondence with similar documents in development process and replies with large quoted text.

This result is a conservative estimate of the amount of redundancy in cloud email traffic, because in practice some messages are read and downloaded multiple times. For example, a Gmail user that reads the same attachment for 10 times, directly from the web browser, generates 90% redundant traffic.

Our experiments show that in order to derive an efficient PACK redundancy elimination, the chunk level redundancy needs to be applied along long chains. To quantify this phenomenon, we ex-



(a) Linux source: 40 different Linux kernel versions



(b) Email: 1-year Gmail account by month

Figure 5: Traffic volume and detected redundancy

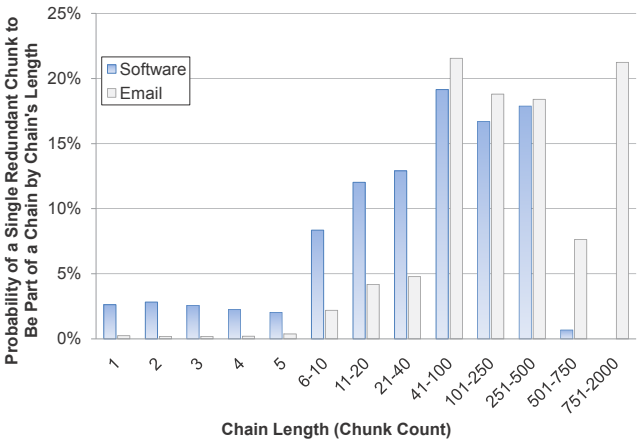


Figure 6: Chain length histogram Linux Software and Email data collections

plored the distribution of redundant chains in the Linux and Email data-sets. Figure 6 presents the resulted redundant data chain length distribution. In Linux 54% of the chunks are found in chains, and in Email about 88%. Moreover, redundant chunks are more probable to reside in long chains. These findings sustain our conclusion that once redundancy is discovered in a single chunk, it is likely to continue in subsequent chunks.

Furthermore, our evaluations show that in videos and large files with a small amount of changes, redundant chunks are likely to reside in very long chains that are efficiently handled by a receiver-based TRE.

5.2 Receiver-Based vs. Sender-Based TRE

In this subsection, we evaluate the sender performance of PACK as well as of a sender-based end-to-end TRE.

5.2.1 *The Server Computational Effort*

First, we evaluate the computational effort of the server in both cases. In PACK, the server is required to perform a SHA-1 operation over a defined range of bytes (the prediction determines a starting point, i.e., offset, and the size of the prediction) only after it verifies that the hint, sent as a part of the prediction, matches the data. In the sender-based TRE, the server is required to first compute Rabin fingerprints in order to slice the stream into chunks, and then to compute a SHA-1 signature for each chunk, prior to sending it. Table 2 presents a summary of the server computational effort of each sender-based TRE described in the literature, as well as of PACK.

To further evaluate the server computational effort for the different sender-based and PACK TRE schemes, we measured the server effort as a function of time and traffic redundancy. For the sender-based scheme we simulated the approach of [12] using their published performance benchmarks[4]. We then measured the server performance as a function of the download time and redundant traffic for the Email data-set, that contains 31.6% redundancy. The sender effort is expressed by the number of SHA-1 operations per second.

Figure 7a demonstrates the high effort placed on a server in a sender-based scheme, compared to the much lower effort of a PACK sender, which performs SHA-1 operations only for data that matches the hint. Moreover, Figure 7b shows that the PACK server computational effort grows linearly with the amount of redundant data. As a result, the server works only when a redundancy is observed and the client reads the data from its local storage instead of receiving it from the server. This scenario demonstrates how the server's and the client's incentives meet: while the server invests the effort into saving traffic volume, the client cooperates to save volume and get faster downloads.

5.2.2 *Synchronization*

Several sender-based end-to-end TRE mechanisms require full synchronization between the sender and the receiver caches. When such synchronization exists, the redundancy is detected and eliminated upfront by the sender. While this synchronization saves an otherwise required three-way handshake, it ignores redundant chunks that arrive at the receiver from different senders. This problem is avoided in PACK, but we did not account this extra efficiency in our current study.

To further understand how TRE would work for a cloud-based web service with returning end-users, we obtained a traffic log from a *social network* site for a period of 33 days at the end of 2010. The data log enables a reliable long term detection of returning users, as users identify themselves using a login to enter the site. We

[4]The taken benchmarks: For 8 KB chunks the SHA-1 calculation throughput is about 250 Mbps with a Pentium III 850MHz and 500 Mbps with a Pentium D 2.8 GHz. Rabin fingerprint chunking is reported to be 2-3 times slower.

Table 2: Sender computational effort comparison between different TRE mechanisms

System	Avg. Chunk Size	Sender Chunking	Sender Signing	Receiver Chunking	Receiver Signing
PACK	Unlimited, receiver's choice	None	SHA-1: true predictions and 0.4% of false predictions	PACK chunking: all real data	SHA-1: all real data
Wanax [12]	Multiple	Multi-Resolution Chunking (MRC): all data	SHA-1: all data	Multi-Resolution Chunking (MRC): all data	SHA-1: all real data
LBFS [20]	Flexible, 8KB	Rabin: all modified data (not relying on database integrity)	SHA-1: all modified data (not relying on database integrity)	Rabin: all modified data (not relying on database integrity)	SHA-1: all modified data (not relying on database integrity)
[3]	None (representative fingerprints)	Rabin: all data (for lookup)	(see Sender Chunking)	None	None
EndRE [1] Chunk-Match	Limited, 32-64 bytes	SampleByte: all data (optionally less, at the cost of reduced compression)	SHA-1: all data (for chunk lookup)	None	None

identified the sessions of 7,000 registered users over this period. We then measured the amount of TRE that can be obtained with different cache sizes at the receiver (a synchronized sender-based TRE keeps a mirror of the last period cache size).

Figure 8 shows the redundancy that can be obtained for different caching periods. Clearly, a short-term cache cannot identify returning long-term sessions.

5.2.3 Users Mobility

Using the social network data-set presented above, we explored the effect of users' mobility on TRE. We focused on users that connected through 3G cellular networks with many device types (PCs, smartphones, etc.). Users are required to complete a registration progress, in which they enter their unique cellular phone number and get a password through SMS message.

We found that 62.1% of the cellular sessions were conducted by users that also got connected to the site through a non-cellular ISP with the same device. Clearly, TRE solutions that are attached to a specific location or rely on static client IP address cannot exploit this redundancy.

Another related finding was that 67.1% of the cellular sessions used IP addresses that were previously used by others in the same data-set. On non-cellular sessions we found only 2.2% of IP reuse. This one is also a major obstacle for synchronized solutions that require a reliable protocol-independent detection of returning clients.

5.3 Estimated Cloud Cost for YouTube Traffic Traces

As noted before, although TRE reduces cloud traffic costs, the increased server efforts for TRE computation result in increased server-hours cost.

We evaluate here the cloud cost of serving the YouTube videos described in Section 5.1 and compare three setups: without TRE, with PACK and with a sender-based TRE. The cost comparison takes into account server-hours and overall outgoing traffic throughput, while omitting storage costs that we found to be very similar in all the examined setups.

The baseline for this comparison is our measurement of a single video server that outputs up to 350 Mbps to 600 concurrent clients. Given a cloud with an array of such servers, we set the cloud policy to add a server when there is less than 0.25 CPU computation power unemployed in the array. A server is removed from this array if, after its removal, there is at least 0.5 CPU power left unused.

Table 3: Cloud operational cost comparison

	No TRE	PACK	Server-based
Traffic volume	9.1 TB	6.4 TB	6.2 TB
Traffic cost reduction (Figure 4)		30%	32%
Server-hours cost increase (Figure 9)		6.1%	19.0%
Total operational cost	100%	80.6%	83.0%

The sender-based TRE was evaluated only using a server's cost for a SHA-1 operation per every outgoing byte, which is performed in all previously published works that can detect YouTube's long-term redundancy.

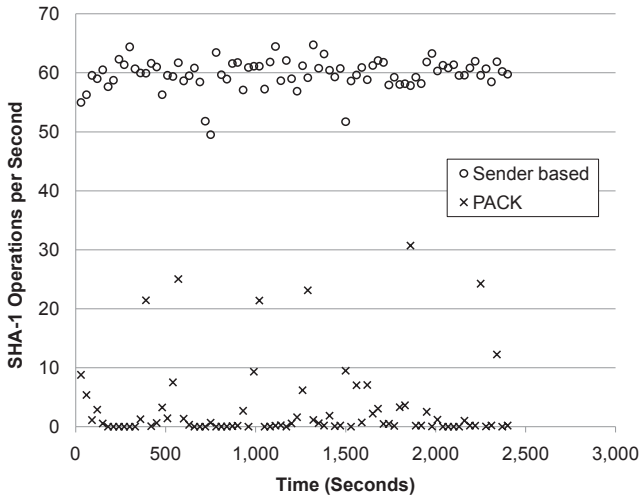
Figure 9 shows, in the shaded part, the number of server-hours used to serve the YouTube traffic from the cloud, with no TRE mechanism. This is our base figure for costs, which is taken as the 100% cost figure for comparison. Figure 9 also shows the number of server-hours needed for this task with either PACK TRE or the sender-based TRE. While PACK puts an extra load of almost one server for only 30% of the time, which accounts for the amount of redundancy eliminated, the sender-based TRE scheme requires between one to two additional servers for almost 90% of the time, resulting in a higher operational cost for 32% redundancy elimination.

Table 3 summarizes the costs and the benefits of the TRE operations and compares them to a baseline with no TRE. The total operational cost is based on current Amazon EC2 [2] pricing for the given traffic-intensive scenario (traffic:server-hours cost ratio of 7:3). Both TRE schemes identify and eliminate the traffic redundancy. However, while PACK employs the server only when redundancy exists, the sender-based TRE employs it for the entire period of time, consuming more servers than PACK and no-TRE schemes when no or little redundancy is detected.

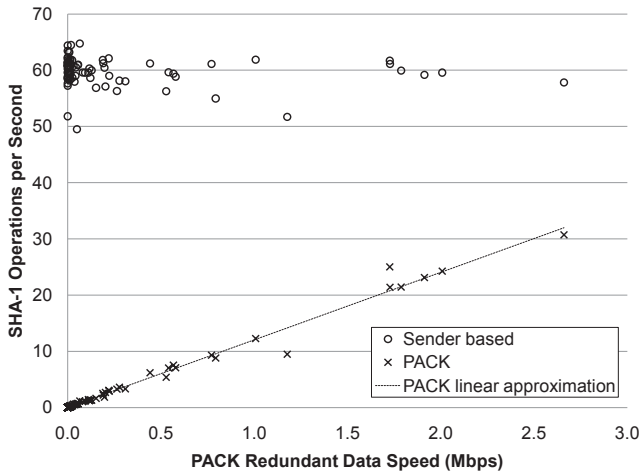
6. IMPLEMENTATION

In this section, we present PACK implementation, its performance analysis and the projected server costs derived from the implementation experiments.

Our implementation contains over 25,000 lines of C and Java



(a) Server effort as a function of time



(b) Sender effort relative to redundant chunks signatures download time (virtual speed)

Figure 7: Difference in computation efforts between receiver and sender-driven modes for the transmission of Email data collection

code. It runs on Linux with Netfilter Queue [21]. Figure 10 shows the PACK implementation architecture. At the server side, we use an Intel Core 2 Duo 3 GHz, 2 GB of RAM and a WD1600AAJS SATA drive desktop. The clients laptop machines are based on an Intel Core 2 Duo 2.8 GHz, 3.5 GB of RAM and a WD2500BJKT SATA drive.

Our implementation enables the transparent use of the TRE at both the server and the client. PACK receiver-sender protocol is embedded in the TCP Options field for low overhead and compatibility with legacy systems along the path. We keep the genuine operating systems' TCP stacks intact, allowing a seamless integration with all applications and protocols above TCP.

Chunking and indexing are performed only at the client's side, enabling the clients to decide independently on their preferred chunk size. In our implementation, the client uses an average chunk size of 8 KB. We found this size to achieve high TRE hit-ratio in the evaluated data-sets, while adding only negligible overheads of 0.1% in meta-data storage and 0.15% in predictions bandwidth.

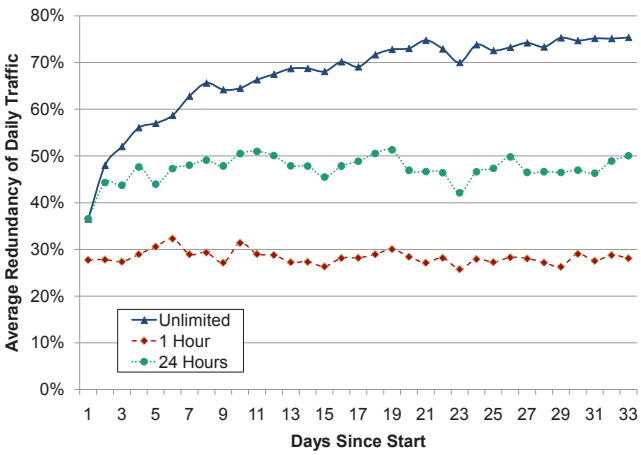


Figure 8: Social network site: traffic redundancy per day with different time-lengths of cache

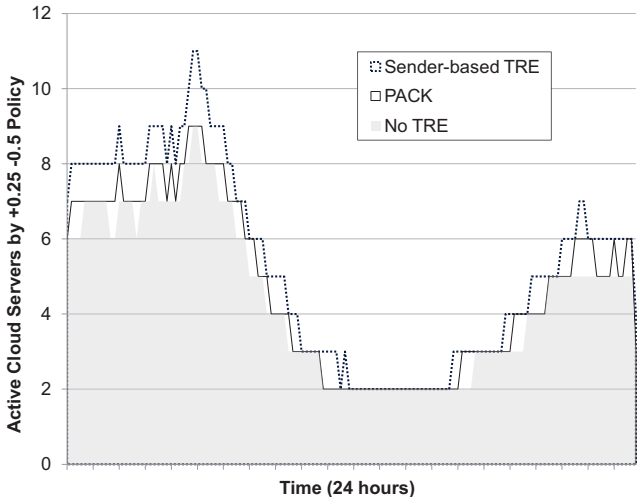


Figure 9: Number of cloud servers needed for serving YouTube traffic without TRE, with sender-based TRE or PACK

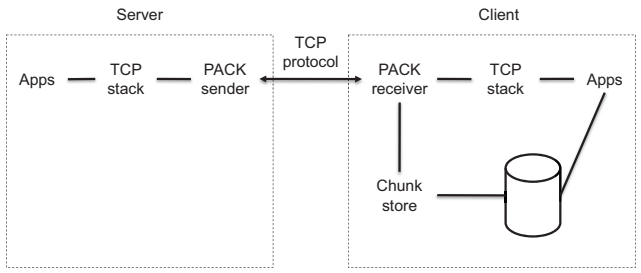


Figure 10: Overview of the PACK implementation

For the experiments held in this section, we generated a workload consisting of Section 5 data-sets: IMAP emails, HTTP videos and files downloaded over FTP. The workload was then loaded to the server, and consumed by the clients. We sampled the machines' status every second to measure real and virtual traffic volumes and CPU utilization.

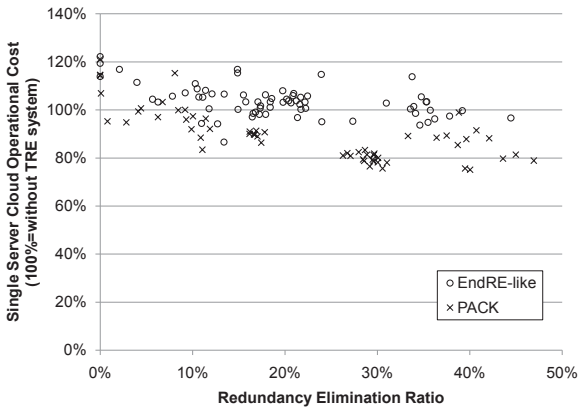


Figure 11: PACK vs. EndRE-like cloud server operational cost as a function of redundancy ratio

6.1 Server Operational Cost

We measured the server performance and cost as a function of the data redundancy level in order to capture the effect of the TRE mechanisms in real environment. To isolate the TRE operational cost, we measured the server's traffic volume and CPU utilization at maximal throughput without operating a TRE. We then used these numbers as a reference cost, based on present Amazon EC2 [2] pricing. The server operational cost is composed of both the network traffic volume and the CPU utilization, as derived from the EC2 pricing.

We constructed a system consisting of one server and seven clients over a 1 Gbps network. The server was configured to provide a maximal throughput of 50 Mbps per client. We then measured three different scenarios, a baseline no-TRE operation, PACK and a sender-based TRE similar to EndRE's Chunk-Match [1], referred to as *EndRE-like*. For the EndRE-like case, we accounted for the SHA-1 calculated over the entire outgoing traffic, but did not account for the chunking effort. In the case of EndRE-like, we made the assumption of unlimited buffers at both the server and client sides to enable the same long-term redundancy level and TRE ratio of PACK.

Figure 11 presents the overall processing and networking cost for traffic redundancy, relative to no-TRE operation. As the redundancy grows, the PACK server cost decreases due to the bandwidth saved by unsent data. However, the EndRE-like server does not gain a significant cost reduction since the SHA-1 operations are performed over non-redundant data too. Note that at above 25% redundancy, which is common to all reviewed data-sets, the PACK operational cost is at least 20% lower than that of EndRE-like.

6.2 PACK Impact on the Client CPU

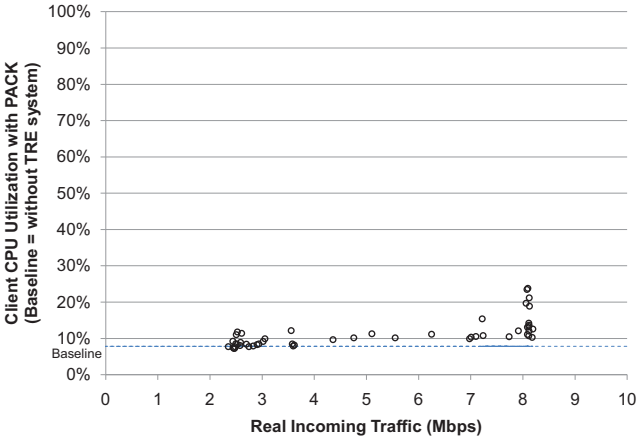
To evaluate the CPU effort imposed by PACK on a client, we measured a random client under a scenario similar to the one used for measuring the server's cost, only this time the cloud server streamed videos at a rate of 9 Mbps to each client. Such a speed throttling is very common in real-time video servers that aim to provide all clients with stable bandwidth for smooth view.

Table 4 summarizes the results. The average PACK-related CPU consumption of a client is less than 4% for 9 Mbps video with 36.4% redundancy.

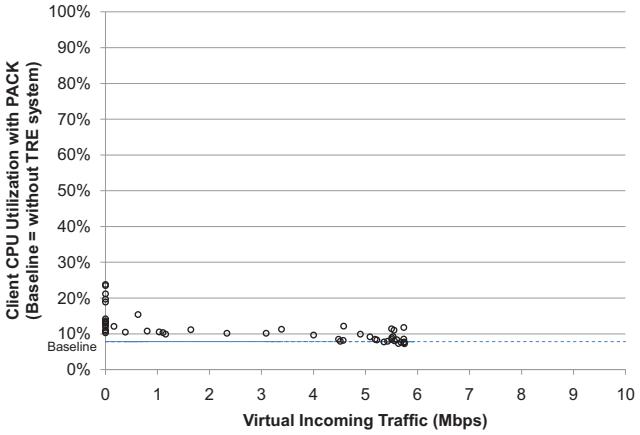
Figure 12a presents the client CPU utilization as a function of the real incoming traffic bandwidth. Since the client chunks the arriving data, the CPU utilization grows as more real traffic enters the client's machine. Figure 12b shows the client CPU utilization

Table 4: Client CPU utilization when streaming 9 Mbps video with and without PACK

No-TRE avg. CPU	7.85%
PACK avg. CPU	11.22%
PACK avg. TRE ratio	36.4%
PACK min CPU	7.25%
PACK max CPU	23.83%



(a) Real traffic



(b) Virtual traffic

Figure 12: Client CPU utilization as a function of the received traffic, when the client's CPU utilization without TRE is used as a baseline

as a function of the virtual traffic bandwidth. Virtual traffic arrives in the form of prediction approvals from the sender, and is limited to a rate of 9 Mbps by the server's throttling. The approvals save the client the need to chunk data or sign the chunks, and enable him to send more predictions based on the same chain that was just used successfully. Hence, the more redundancy is found, the less CPU utilization incurred by PACK.

6.3 Chunking Scheme

Our implementation employs a novel computationally light-weight chunking (fingerprinting) scheme, termed *PACK chunking*. The scheme, presented in Proc. 8 and illustrated in Figure 13, is a XOR-

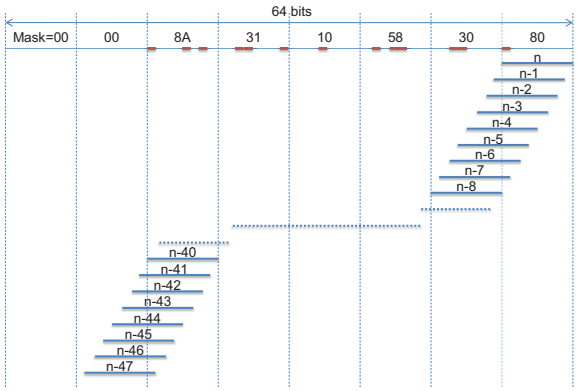


Figure 13: PACK chunking: a snapshot after at least 48 bytes were processed

Table 5: Chunking schemes processing speed tested with 10 MB random file over a client's laptop, without neither minimal nor maximal limit on the chunk size.

Scheme	Window	Chunks	Speed
SampleByte 8 markers	1 byte	32 bytes	1,913 Mbps
Rabin fingerprint	48 bytes	8 KB	2,686 Mbps
PACK chunking	48 bytes	8 KB	3,259 Mbps
SampleByte 1 marker	1 byte	256 bytes	5,176 Mbps

based rolling hash function, tailored for fast TRE chunking. Anchors are detected by the mask in line 1 that provides on average 8 KB chunks while considering all the 48 bytes in the sliding window.

Our measurements show that *PACK chunking* is faster than the fastest known Rabin fingerprint software implementation [8], due to a one less XOR operation per byte.

Proc. 8 PACK chunking algorithm

1. $mask \Leftarrow 0x00008A3110583080$ {48 bytes window; 8 KB chunks}
2. $longval \Leftarrow 0$ {has to be 64 bits}
3. **for all** byte $\in$ stream **do**
4. shift left $longval$ by 1 bit {lsb $\leftarrow$ 0; drop msb}
5. $longval \Leftarrow longval$ bitwise-xor $byte$
6. **if** processed at least 48 bytes **and** ($longval$ bitwise-and $mask$) == $mask$ **then**
7. found an anchor
8. **end if**
9. **end for**

We further measured *PACK chunking* speed and compared it to other schemes. The measurements were performed on an unloaded CPU whose only operation was to chunk a 10 MB random binary file. Table 5 summaries the processing speed of the different chunking schemes. As a baseline figure we measured the speed of SHA-1 signing, and found that it reached 946 Mbps.

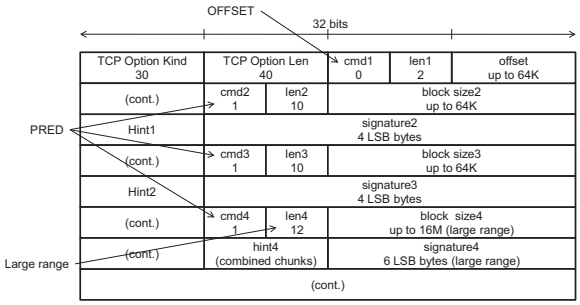


Figure 14: Receiver message example of a large range prediction

6.4 PACK Messages Format

In our implementation, we use two currently unused TCP option codes, similar to the ones defined in SACK [17]. The first one is an enabling option *PACK permitted* sent in a SYN segment to indicate that the PACK option can be used after the connection is established. The other one is a *PACK message* that may be sent over an established connection once permission has been granted by both parties. A single PACK message, piggybacked on a single TCP packet, is designed to wrap and carry multiple PACK commands, as illustrated in Figure 14. This not only saves message overhead, but also copes with security network devices (e.g. firewall) that tend to change TCP options order [19]. Note, that most TCP options are only used at the TCP initialization period, with several exceptions such as SACK [17] and timestamps [13][19]. Due to the lack of space, additional implementation details are left out and are available in [22].

7. CONCLUSIONS

Cloud computing is expected to trigger high demand for TRE solutions as the amount of data exchanged between the cloud and its users is expected to dramatically increase. The cloud environment redefines the TRE system requirements, making proprietary middle-box solutions inadequate. Consequently, there is a rising need for a TRE solution that reduces the cloud's operational cost, while accounting for application latencies, user mobility and cloud elasticity.

In this work, we have presented PACK, a receiver-based, cloud friendly end-to-end TRE which is based on novel speculative principles that reduce latency and cloud operational cost. PACK does not require the server to continuously maintain clients' status, thus enabling cloud elasticity and user mobility while preserving long-term redundancy. Besides, PACK is capable of eliminating redundancy based on content arriving to the client from multiple servers without applying a three-way handshake.

Our evaluation using a wide collection of content types shows that PACK meets the expected design goals and has clear advantages over sender-based TRE, especially when the cloud computation cost and buffering requirements are important. Moreover, PACK imposes additional effort on the sender only when redundancy is exploited, thus reducing the cloud overall cost.

Two interesting future extensions can provide additional benefits to the PACK concept. First, our implementation maintains chains by keeping for any chunk only the last observed subsequent chunk in a LRU fashion. An interesting extension to this work is the statistical study of chains of chunks that would enable multiple possibilities in both the chunk order and the corresponding predictions. The system may also allow making more than one prediction at a time and it is enough that one of them will be correct for successful

traffic elimination. A second promising direction is the mode of operation optimization of the hybrid sender-receiver approach based on shared decisions derived from receiver's power or server's cost changes.

8. ACKNOWLEDGEMENTS

This work is partially supported by the VENUS-C project, co-funded within the 7th Framework Programme by the GÉANT & e-Infrastructure Unit, Information Society & Media Directorate General of the European Commission, Contract 261565. It is also partially supported by NET-HD MAGNET program of the office of the Chief Scientist of the Israeli Ministry of Industry, Trade, and Labor. We would like to thank our partners in Venus-C and NET-HD projects for their invaluable feedback and insight.

The authors would like to thank the anonymous SIGCOMM 2011 reviewers and our shepherd, Aditya Akella, for their comments and suggestions that considerably helped us to improve the final version.

9. REFERENCES

[1] 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 *Proc. of NSDI*, 2010.

[2] Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2/`.

[3] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet caches on routers: The implications of universal redundant traffic elimination. In *Proc. of SIGCOMM*, pages 219–230, New York, NY, USA, 2008. ACM.

[4] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee. Redundancy in Network Traffic: Findings and Implications. In *Proc. of SIGMETRICS*, pages 37–48. ACM New York, NY, USA, 2009.

[5] A. Anand, V. Sekar, and A. Akella. SmartRE: an Architecture for Coordinated Network-Wide Redundancy Elimination. In *Proc. of SIGCOMM*, volume 39, pages 87–98, New York, NY, USA, 2009. ACM.

[6] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[7] BlueCoat Systems. `http://www.bluecoat.com/`, 1996.

[8] A. Z. Broder. Some Applications of Rabin's Fingerprinting Method. In *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[9] Expand Networks: Application Acceleration and WAN Optimization. `http://www.expand.com/technology/application-acceleration.aspx`, 1998.

[10] F5: WAN Optimization. `http://www.f5.com/solutions/acceleration/wan-optimization/`, 1996.

[11] A. Gupta, A. Akella, S. Seshan, S. Shenker, and J. Wang. Understanding and Exploiting Network Traffic Redundancy. Technical Report 1592, UW-Madison, April 2007.

[12] S. Ihm, K. Park, and V. Pai. Wide-area Network Acceleration for the Developing World. In *Proc. of USENIX ATC*, pages 18–18, Berkeley, CA, USA, 2010. USENIX Association.

[13] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. *RFC 1323*, 1992.

[14] Juniper Networks: Application Acceleration. `http://www.juniper.net/us/en/products-services/application-acceleration/`, 1996.

[15] E. Lev-Ran, I. Cidon, and I. Z. Ben-Shaul. Method and Apparatus for Reducing Network Traffic over Low Bandwidth Links. *US Patent 7636767*, November 2009. Filed: November 2005.

[16] U. Manber. Finding similar files in a large file system. In *Proc. of the USENIX winter technical conference*, pages 1–10, Berkeley, CA, USA, 1994. USENIX Association.

[17] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. *RFC 2018*, 1996.

[18] S. Mccanne and M. Demmer. Content-Based Segmentation Scheme for Data Compression in Storage and Transmission Including Hierarchical Segment Representation. *US Patent 6828925*, December 2004. Filed: December 2003.

[19] A. Medina, M. Allman, and S. Floyd. Measuring the evolution of transport protocols in the Internet. *ACM Computer Communication Review*, 35(2):37–52, 2005.

[20] A. Muthitacharoen, B. Chen, and D. Mazi'eres. A Low-Bandwidth Network File System. In *Proc. of SOSP*, pages 174–187, New York, NY, USA, 2001. ACM.

[21] netfilter/iptables project: libnetfilter_queue. `http://www.netfilter.org/projects/libnetfilter_queue`, Oct 2005.

[22] PACK source code. `http://www.venus-c.eu/pages/partner.aspx?id=10`.

[23] N. T. Spring and D. Wetherall. A Protocol-Independent Technique for Eliminating Redundant Network Traffic. In *Proc. of SIGCOMM*, volume 30, pages 87–95, New York, NY, USA, 2000. ACM.

[24] R. Williams. Method for Partitioning a Block of Data Into Subblocks and for Storing and Communcating Such Subblocks. *US Patent 5990810*, November 1999. Filed: August 1996.

[25] M. Zink, K. Suh, Y. Gu, and J. Kurose. Watch Global, Cache Local: YouTube Network Traffic at a Campus Network - Measurements and Implications. In *Proc. of MMCN*, San Jose, CA, USA, 2008.