PACK: Prediction-Based Cloud Bandwidth and Cost Reduction System

Eyal Zohar, Israel Cidon, and Osnat Mokryn

***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.**

***Index Terms*—Caching, cloud computing, network optimization, traffic redundancy elimination.**

I. Introduction

CLOUD computing offers its customers an economical and convenient *pay-as-you-go* service model, known also as *usage-based pricing* [2]. 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. In particular, data transfer costs (i.e., bandwidth) is an important issue when trying to minimize costs [2]. 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, uploading (i.e., backup), 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 [3]–[5]. 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 [6], Riverbed [7], Quantum [8], Juniper [9], Blue Coat [10], Expand Networks [11], and F5 [12]).

While proprietary middle-boxes are popular point solutions within enterprises, they are not as attractive in a cloud environment. Cloud providers cannot benefit from a technology whose goal is to reduce customer bandwidth bills, and thus are not likely to invest in one. The rise of "on-demand" work spaces, meeting rooms, and work-from-home solutions [13] detaches the workers from their offices. In such a dynamic work environment, fixed-point solutions that require a client-side and a server-side middle-box pair become ineffective. On the other hand, cloud-side elasticity motivates work distribution among servers 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 [14], [15]. This enables the use of a standard protocol stack and makes a TRE within end-to-end secured traffic (e.g., SSL) possible.

Current end-to-end TRE solutions are sender-based. In the case where the cloud server is the sender, these solutions require that the server continuously maintain clients' status.

We show here that *cloud elasticity* calls for a new TRE solution. 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 efficiency due to lost synchronization. Second, the popularity of rich media that consume high bandwidth motivates content distribution network (CDN) solutions, in which the service point for fixed and mobile users may change dynamically according to the relative service point locations and loads. Moreover, 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.

Manuscript received May 01, 2012; revised December 01, 2012; accepted January 06, 2013; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor L. E. Li.

E. Zohar and I. Cidon are with the Department of Electrical Engineering, Technion—Israel Institute of Technology, Haifa 44444, Israel (e-mail: eyalzo@tx.technion.ac.il; cidon@ee.technion.ac.il).

O. Mokryn is with the Computer Science Department, Tel Aviv College, Tel-Aviv 46410, Israel (e-mail: ossi@mta.ac.il).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2013.2240010

[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 services.

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 [4], [15] add a considerable load to the servers, which may eradicate the cloud cost saving addressed by the TRE in the first place. 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' metadata 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. The sender first examines the hint and performs the TRE operation only on a hint-match. The purpose of this procedure is to avoid the expensive TRE computation at the sender side in the absence of traffic redundancy. When redundancy is detected, the sender then sends to the receiver only the ACKs to the predictions, instead of sending the data.

On the receiver side, we propose a new computationally lightweight 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 Gb/s, at least 20% faster than Rabin fingerprinting.

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 [15].

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 [16]. Our implementation utilizes the TCP Options field, supporting all TCP-based applications such as Web, video streaming, P2P, e-mail, etc.

In addition, we evaluate our solution and compare it to previous end-to-end solutions using terabytes 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.

This paper is organized as follows. Section II reviews existing TRE solutions. In Section III, we present our receiver-based TRE solution and explain the prediction process and the prediction-based TRE mechanism. In Section IV, we present optimizations to the receiver-side algorithms. Section V evaluates data redundancy in a cloud and compares PACK to sender-based TRE. Section VI details our implementation and discusses our experiments and results.

II. Related Work

Several TRE techniques have been explored in recent years. A protocol-independent TRE was proposed in [4]. The paper describes a packet-level TRE, utilizing the algorithms presented in [3].

Several commercial TRE solutions described in [6] and [7] have combined the sender-based TRE ideas of [4] with the algorithmic and implementation approach of [5] along with protocol specific optimizations for middle-boxes solutions. In particular, [6] describes how to get away with three-way handshake between the sender and the receiver if a full state synchronization is maintained.

References [17] and [18] 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 [19], [20], and [14]. In the latter, packet-level TRE techniques are compared [3], [21]. Our paper builds on their 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 [22] 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 [15] 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 B. 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.

[2]We generally assume that the cloud side, following the current Web service model, is dominated by a sender operation. The cases where the cloud is the receiver are referenced specifically.

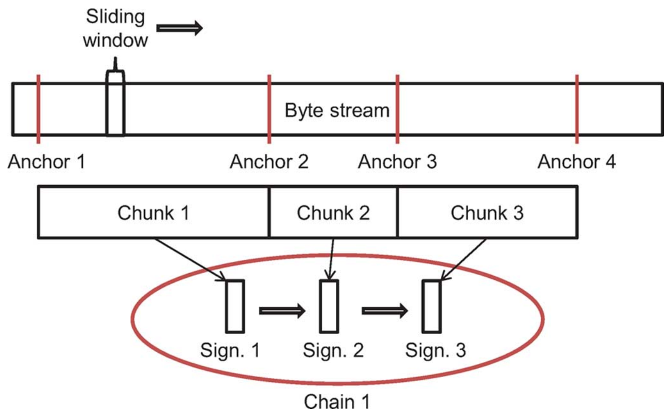


Fig. 1. From stream to chain.

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.

III. 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 IV.

The stream of data received at the PACK receiver is parsed to a sequence of variable-size, content-based signed chunks similar to [3] and [5]. 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 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 by 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.

A. Receiver Chunk Store

PACK uses a new *chains* scheme, described in Fig. 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 metadata. Chunk's metadata 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 are 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 metadata 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 metadata 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.

The utilization of a small chunk size presents better redundancy elimination when data modifications are fine-grained, such as sporadic changes in an 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.

Unlike IP-level TRE solutions that are limited by the IP packet size ($\sim$ 1500 B), 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 8 kB (see Section VI).

B. 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' metadata. 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
```

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

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
```

C. Sender Algorithm

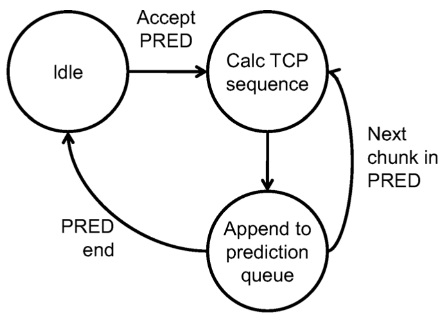
When a sender receives a PRED message from the receiver, it tries to match the received predictions to its buffered (yet to be sent) data. For each prediction, the sender determines the corresponding TCP sequence range and verifies the hint. Upon a hint match, the sender calculates the more computationally 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.

Fig. 2 illustrates the sender operation using state machines. Fig. 2(a) describes the parsing of a received PRED command. Fig. 2(b) 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.

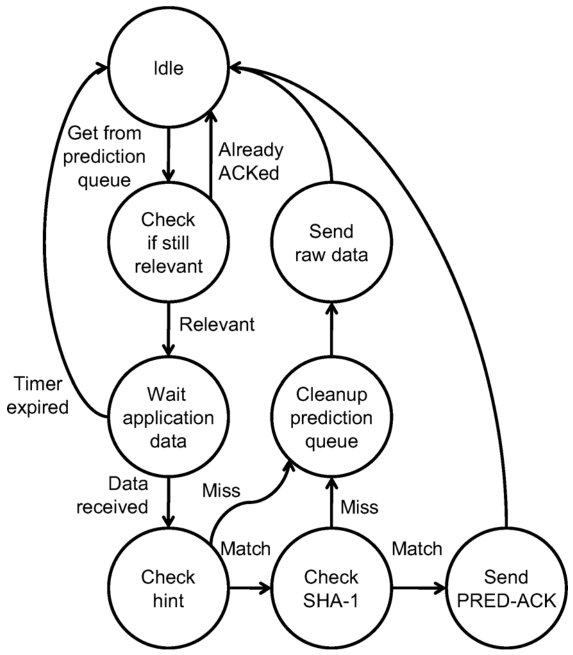
D. Wire Protocol

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

Fig. 3 illustrates the way the PACK wire 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.



(a)



(b)

Fig. 2. Sender algorithms. (a) Filling the prediction queue. (b) Processing the prediction queue and sending PRED-ACK or raw data.

IV. Optimizations

For the sake of clarity, Section III presents the most basic version of the PACK protocol. In this section, we describe additional options and optimizations.

A. 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

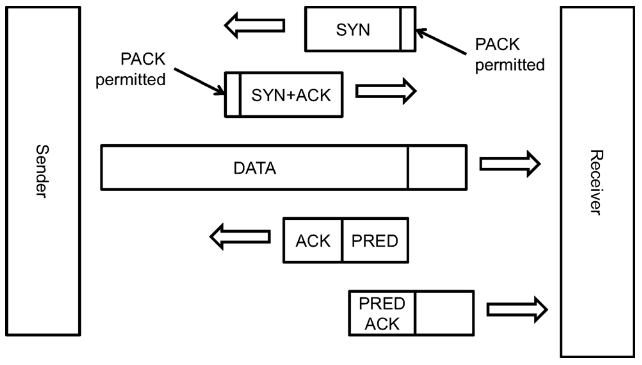


Fig. 3. PACK wire protocol in a nutshell.

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 an 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 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.

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()
```

The size increase of the virtual window introduces a tradeoff 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 is found in the chunk store. Note that even a slight change in the sender's data, compared to 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.

B. Cloud Server as a Receiver

In a growing trend, cloud storage is becoming a dominant player [23], [24]—from backup and sharing services [25] to the American National Library [26], and e-mail services [27], [28]. In many of these services, the cloud is often the receiver of the data.

If the sending client has no power limitations, PACK can work to save bandwidth on the upstream to the cloud. In these cases, the end-user acts as a sender, and the cloud server is the receiver. The PACK algorithm need not change. It does require, however, that the cloud server—like any PACK receiver—maintain a *chunk store*.

C. 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, described in Proc. 6 and Proc. 7. When PACK recognizes a pattern of dispersed changes, it may select to trigger a sender-driven approach in the spirit of [4], [6], [7], and [18].

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

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.

TABLE I
DATA AND PACK'S RESULTS OF 24 hYOUTUBE 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

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.

V. MOTIVATING A RECEIVER-BASED APPROACH

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: 1) video traces captured at a major ISP; 2) traffic obtained from a popular social network service; and 3) 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.

A. Traffic Redundancy

1) Traffic Traces: We obtained a 24-h recording of traffic at an ISP's 10-Gb/s PoP router, using a 2.4-GHz CPU recording machine with 2 TB storage (4×500 GB 7 200 RPM disks) and 1-Gb/s 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. For accurate reading of the true redundancy, we 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 I summarizes our findings. We recorded more than 146 K distinct sessions, in which 37 K users request over 39 K distinct movies. Average movie size is 15 MB, while the

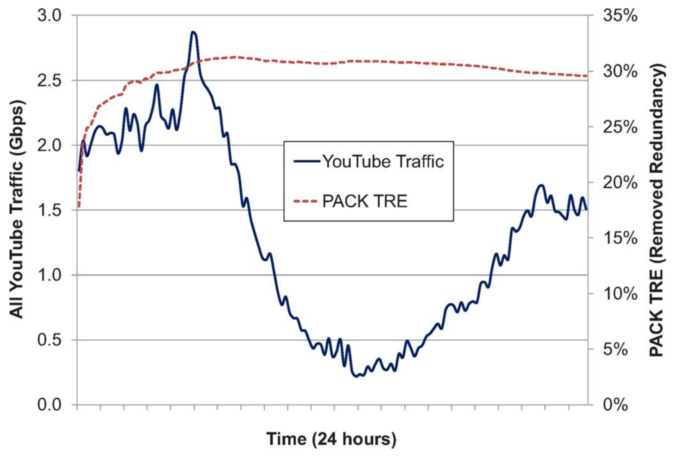


Fig. 4. ISP's YouTube traffic over 24 h, and PACK redundancy elimination ratio with this data.

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.

Fig. 4 presents the YouTube traffic and the redundancy obtained by PACK over the entire period, with the redundancy sampled every 10 min 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 intersession 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 h, which results in some replays that do not produce downloads at all.

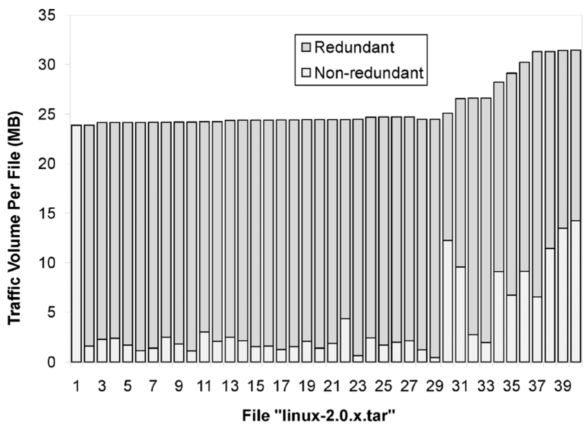
2) Static Dataset: We acquired the following static datasets:

Linux source—different Linux kernel versions: all the 40 2.0.x tar files of the kernel source code that sum up to 1 GB;

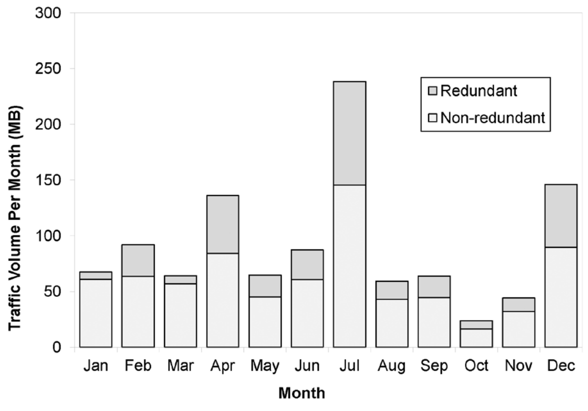
Email—a single-user Gmail account with 1140 e-mail messages over a year that sum up to 1.09 GB.

The 40 Linux source versions were released over a period of 2 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. Fig. 5(a) 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 e-mail traffic, we operated an IMAP client that fully synchronized the remote Gmail account with a new local folder. Fig. 5(b) shows the redundancy in each month, according to the e-mail 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, e-mail correspondence with similar documents in development process, and replies with large quoted text.



(a)



(b)

Fig. 5. Traffic volume and detected redundancy. (a) Linux source: 40 different Linux kernel versions. (b) Email: 1-year Gmail account by month.

This result is a conservative estimate of the amount of redundancy in cloud e-mail 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 explored the distribution of redundant chains in the Linux and Email datasets. Fig. 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.

TABLE II
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 [22]	Multiple	Multi-Resolution Chunking (MRC): all data	SHA-1: all data	Multi-Resolution Chunking (MRC): all data	SHA-1: all real data
LBFS [5]	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)
[17]	None (representative fingerprints)	Rabin: all data (for lookup)	(see Sender Chunking)	None	None
EndRE [15] 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

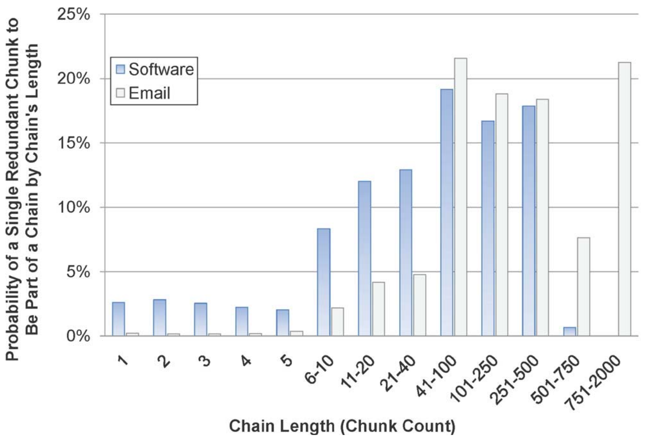


Fig. 6. Chain length histogram Linux Software and Email data collections.

B. *Receiver-Based Versus Sender-Based TRE*

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

1) Server Computational Effort: First, we evaluate the computational effort of the server in both cases. In PACK, the server is required to perform an 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 an SHA-1 signature for each chunk, prior to sending it. Table II 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 [22] 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 dataset, which contains 31.6% redundancy. The sender effort is expressed by the number of SHA-1 operations per second.

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

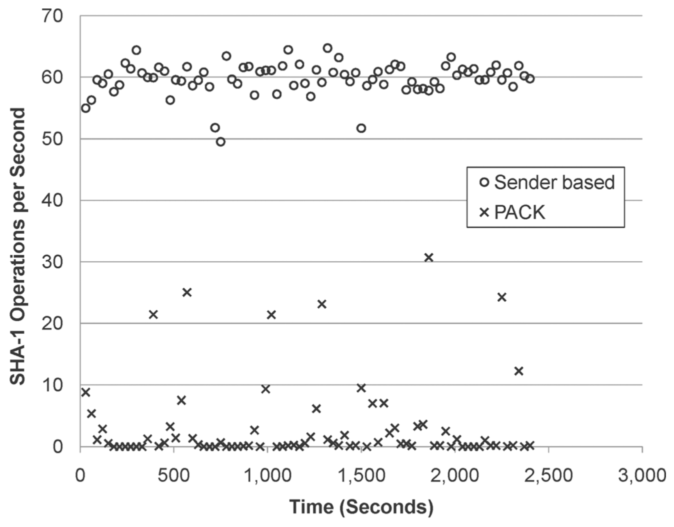
Fig. 7(a) demonstrates the high effort placed on a server in a sender-based scheme, compared to the much lower effort of a PACK sender, which performs SHA-1 operations only for data that matches the hint. Moreover, Fig. 7(b) shows that the PACK server computational effort grows linearly with the amount 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.

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 up front 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.

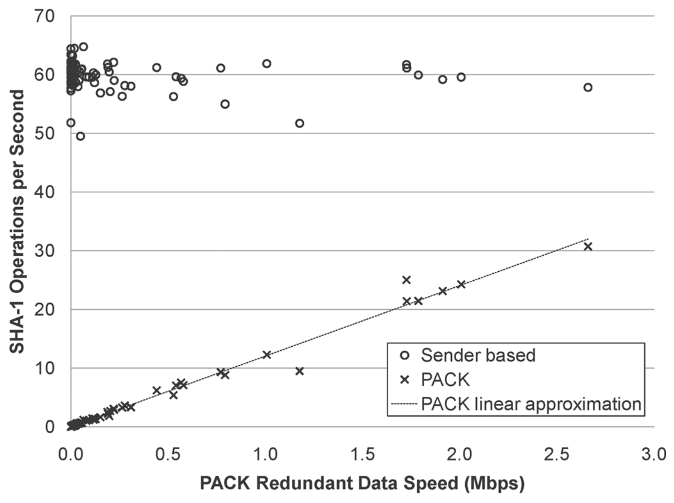
To further understand how TRE would work for a cloud-based Web service with returning end-users, we obtained a traffic log from a *social network* site for a period of 33 days at the end of 2010. The data log enables a reliable long-term detection of returning users, as users identify themselves using a login to enter the site. We identified the sessions of 7000 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).

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

3) Users Mobility: Using the social network dataset presented above, we explored the effect of users' mobility on TRE. Clearly, PACK is not directed to mobile devices, but to users who use their stations from different locations. However, in this experiment 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



(a)



(b)

Fig. 7. Difference in computation efforts between receiver and sender-driven modes for the transmission of Email data collection. (a) Server effort as a function of time. (b) Sender effort relative to redundant chunks signatures download time (virtual speed).

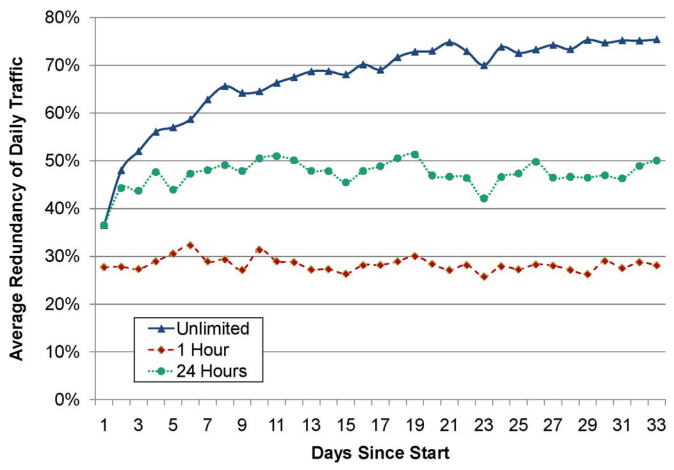


Fig. 8. Social network site: traffic redundancy per day with different time lengths of cache.

they enter their unique cellular phone number and get a password through SMS message.

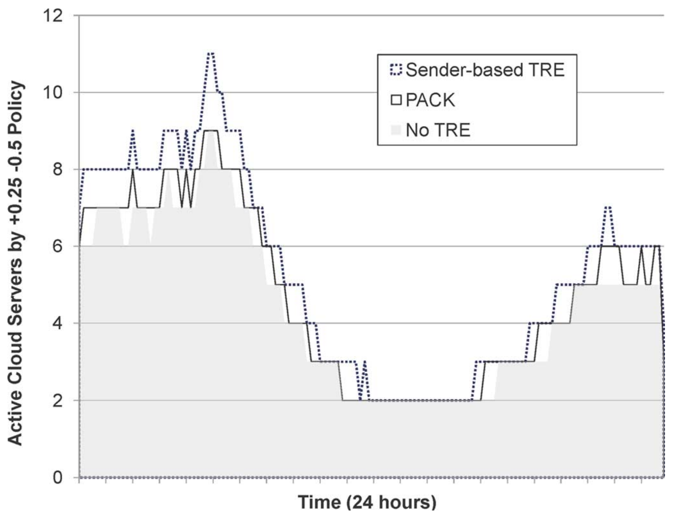


Fig. 9. Number of cloud servers needed for serving YouTube traffic without TRE, with sender-based TRE, or with PACK.

We found that 62.1% of the cellular sessions were conducted by users that also got connected to the site through a noncellular 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 dataset. On noncellular 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.

C. *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 V-A 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 Mb/s 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.

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

Fig. 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. Fig. 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

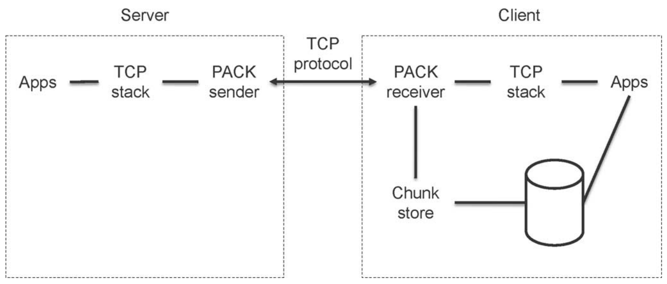


Fig. 10. Overview of the PACK implementation.

TABLE III
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%

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 III 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 [29] 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.

VI. 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 code. It runs on Linux with Netfilter Queue [30]. Fig. 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 datasets, while adding only negligible overheads of 0.1% in metadata storage and 0.15% in predictions bandwidth.

For the experiments held in this section, we generated a workload consisting of Section V datasets: IMAP e-mails, 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.

A. 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 [29] 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-Gb/s network. The server was configured to provide a maximal throughput of 50 Mb/s 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 [15], 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.

Fig. 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 nonredundant data as well. Note that at above 25% redundancy, which is common to all reviewed datasets, the PACK operational cost is at least 20% lower than that of EndRE-like.

B. 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 Mb/s 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 IV summarizes the results. The average PACK-related CPU consumption of a client is less than 4% for 9-Mb/s video with 36.4% redundancy.

Fig. 12(a) 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. Fig. 12(b) shows the client CPU utilization as a function of the virtual traffic bandwidth. Virtual traffic arrives in the form of prediction approvals from the

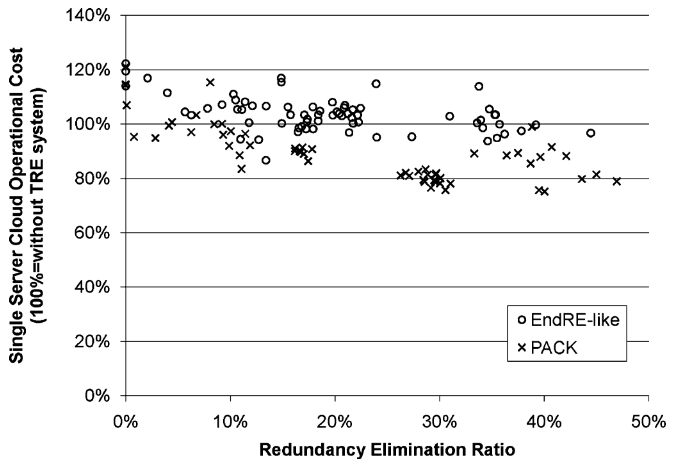
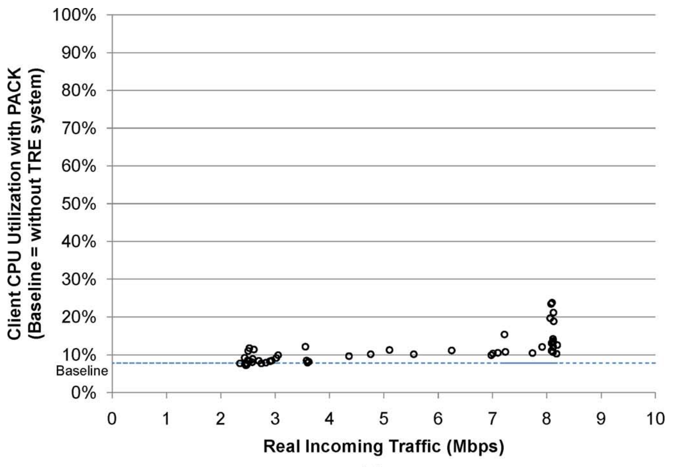


Fig. 11. PACK versus EndRE-like cloud server operational cost as a function of redundancy ratio.

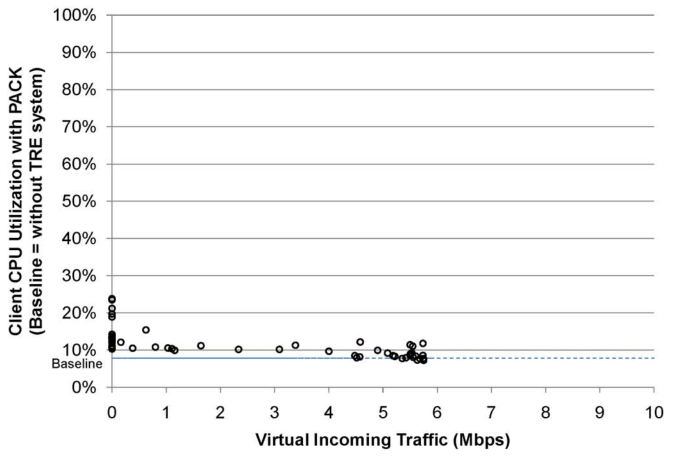
TABLE IV
CLIENT CPU UTILIZATION WHEN STREAMING 9-Mb/s VIDEO WITH AND WITHOUT PACK

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%

sender and is limited to a rate of 9 Mb/s 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.



(a)



(b)

Fig. 12. Client CPU utilization as a function of the received traffic, when the client's CPU utilization without TRE is used as a baseline. (a) Real traffic. (b) Virtual traffic.

C. Chunking Scheme

Our implementation employs a novel computationally lightweight chunking (fingerprinting) scheme, termed *PACK chunking*. The scheme, presented in Proc. 8 and illustrated in Fig. 13, is an XOR-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. The mask, as shown in Fig. 13, was chosen to consider all the 48 B in the sliding window.

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 ∈ stream do
4.     shift left longval by 1 bit {lsb ← 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
```

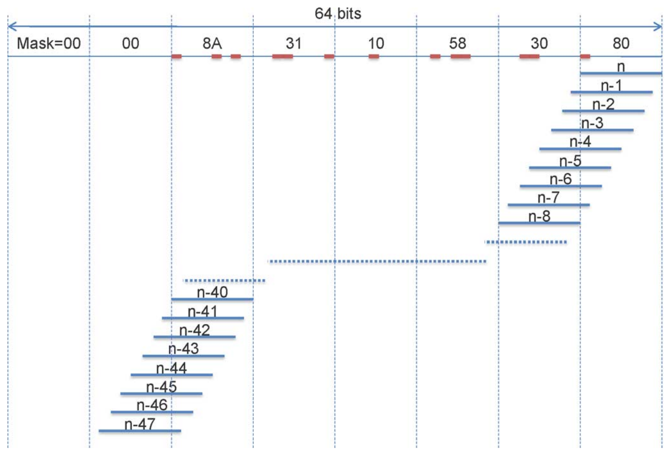


Fig. 13. PACK chunking: snapshot after at least 48 B were processed.

Our measurements show that *PACK chunking* is faster than the fastest known Rabin fingerprint software implementation [31] due to one less XOR operation per byte.

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

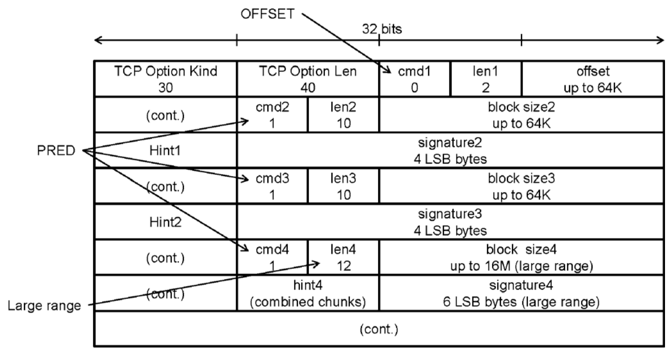


Fig. 14. Receiver message example of a large range prediction.

TABLE V
CHUNKING SCHEMES PROCESSING SPEED TESTED WITH 10-MB RANDOM FILE OVER A CLIENT'S LAPTOP, WITHOUT EITHER MINIMAL OR 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

random binary file. Table V 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 Mb/s.

D. Pack Messages Format

In our implementation, we use two currently unused TCP option codes, similar to the ones defined in SACK [32]. 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 Fig. 14. This not only saves message overhead, but also copes with security network devices (e.g., firewall) that tend to change TCP options order [33]. Note that most TCP options are only used at the TCP initialization period, with several exceptions such as SACK [32] and timestamps [34], [33]. Due to the lack of space, additional implementation details are left out and are available in [16].

VII. CONCLUSION

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 paper, we have presented PACK, a receiver-based, cloud-friendly, end-to-end TRE that 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. Moreover, 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 an 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.

REFERENCES

[1] E. Zohar, I. Cidon, and O. Mokryn, "The power of prediction: Cloud bandwidth and cost reduction," in *Proc. SIGCOMM*, 2011, pp. 86–97.
[2] 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, no. 4, pp. 50–58, 2010.
[3] U. Manber, "Finding similar files in a large file system," in *Proc. USENIX Winter Tech. Conf.*, 1994, pp. 1–10.
[4] N. T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," in *Proc. SIGCOMM*, 2000, vol. 30, pp. 87–95.
[5] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *Proc. SOSP*, 2001, pp. 174–187.
[6] E. Lev-Ran, I. Cidon, and I. Z. Ben-Shaul, "Method and apparatus for reducing network traffic over low bandwidth links," US Patent 7636767, Nov. 2009.
[7] S. Mccanne and M. Demmer, "Content-based segmentation scheme for data compression in storage and transmission including hierarchical segment representation," US Patent 6828925, Dec. 2004.
[8] R. Williams, "Method for partitioning a block of data into subblocks and for storing and communicating such subblocks," US Patent 5990810, Nov. 1999.
[9] Juniper Networks, Sunnyvale, CA, USA, "Application acceleration," 1996 [Online]. Available: http://www.juniper.net/us/en/products-services/application-acceleration/
[10] Blue Coat Systems, Sunnyvale, CA, USA, "MACH5," 1996 [Online]. Available: http://www.bluecoat.com/products/mach5
[11] Expand Networks, Riverbed Technology, San Francisco, CA, USA, "Application acceleration and WAN optimization," 1998 [Online]. Available: http://www.expand.com/technology/application-acceleration.aspx
[12] F5, Seattle, WA, USA, "WAN optimization," 1996 [Online]. Available: http://www.f5.com/solutions/acceleration/wan-optimization/
[13] A. Flint, "The next workplace revolution," Nov. 2012 [Online]. Available: http://m.theatlanticcities.com/jobs-and-economy/2012/11/nextworkplace-revolution/3904/
[14] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee, "Redundancy in network traffic: Findings and implications," in *Proc. SIGMETRICS*, 2009, pp. 37–48.

[15] 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. NSDI*, 2010, pp. 28–28.

[16] “PACK source code,” 2011 [Online]. Available: http://www.eyalzo.com/projects/pack

[17] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker, “Packet caches on routers: The implications of universal redundant traffic elimination,” in *Proc. SIGCOMM*, 2008, pp. 219–230.

[18] A. Anand, V. Sekar, and A. Akella, “SmartRE: An architecture for coordinated network-wide redundancy elimination,” in *Proc. SIGCOMM*, 2009, vol. 39, pp. 87–98.

[19] A. Gupta, A. Akella, S. Seshan, S. Shenker, and J. Wang, “Understanding and exploiting network traffic redundancy,” UW-Madison, Madison, WI, USA, Tech. Rep. 1592, Apr. 2007.

[20] 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. MMCN*, 2008, pp. 1–13.

[21] S. Schleimer, D. S. Wilkerson, and A. Aiken, “Winnowing: Local algorithms for document fingerprinting,” in *Proc. SIGMOD*, 2003, pp. 76–85.

[22] S. Ihm, K. Park, and V. Pai, “Wide-area network acceleration for the developing world,” in *Proc. USENIX ATC*, 2010, pp. 18–18.

[23] H. Stevens and C. Pettey, “Gartner says cloud computing will be as influential as e-business,” *Gartner Newsroom*, Jun. 26, 2008.

[24] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, “RACS: A case for cloud storage diversity,” in *Proc. SOCC*, 2010, pp. 229–240.

[25] “Dropbox,” 2007 [Online]. Available: http://www.dropbox.com/

[26] E. Allen and C. M. Morris, “Library of Congress and Duracloud launch pilot program using cloud technologies to test perpetual access to digital content,” News Release, Jul. 2009.

[27] D. Hansen, “GMail filesystem over FUSE,” [Online]. Available: http://sr71.net/projects/gmailfs/

[28] J. Srinivasan, W. Wei, X. Ma, and T. Yu, “EMFS: Email-based personal cloud storage,” in *Proc. NAS*, 2011, pp. 248–257.

[29] “Amazon Elastic Compute Cloud (EC2),” [Online]. Available: http://aws.amazon.com/ec2/

[30] “netfilter/iptables project: Libnetfilter_queue,” Oct. 2005 [Online]. Available: http://www.netfilter.org/projects/libnetfilter_queue

[31] A. Z. Broder, “Some applications of Rabin’s fingerprinting method,” in *Sequences II: Methods in Communications, Security, and Computer Science*. New York, NY, USA: Springer-Verlag, 1993, pp. 143–152.

[32] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, “TCP selective acknowledgment options,” RFC 2018, 1996.

[33] A. Medina, M. Allman, and S. Floyd, “Measuring the evolution of transport protocols in the internet,” *Comput. Commun. Rev.*, vol. 35, no. 2, pp. 37–52, 2005.

[34] V. Jacobson, R. Braden, and D. Borman, “TCP extensions for high performance,” RFC 1323, 1992.

Eyal Zohar received the B.A. and M.Sc. degrees in computer science from the Open University, Ra’anana, Israel, in 1998 and 2009, respectively, and is currently pursuing the Ph.D. degree in electrical engineering at the Technion—Israel Institute of Technology, Haifa, Israel.

He has a vast experience in network-related software-development from several startup companies. His research interests include P2P networks, content caching, cloud computing, cellular networks, and traffic redundancy elimination.

Mr. Zohar is a recipient of the Intel Award for graduate students in 2012.

Israel Cidon received the B.Sc. and Ph.D. degrees in electrical engineering from the Technion—Israel Institute of Technology, Haifa, Israel, in 1980 and 1984, respectively.

He is a Professor of electrical engineering with the Technion. From 2005 to 2010, he was the Dean of the Electrical Engineering Faculty with the Technion. Between 1985 and 1994, he was the Manager of the Network Architecture and Algorithms with the IBM T. J. Watson Research Center, Yorktown Heights, NY, leading several computer networks projects including the first implementations of a packet-based multimedia network and IBM first storage area network. During 1994 and 1995, he founded and managed the high-speed networking group with Sun Microsystems Labs, Mountain View, CA. In 1981 he cofounded Micronet Ltd., Tel-Aviv, Israel, an early vendor of mobile data-entry terminals. In 1998, he cofounded Viola Networks, Yokneam, Israel, a provider network and VoIP performance diagnostic suite (acquired by Fluke Networks, 2008). In 2000, he cofounded Actona Technologies, Haifa, Israel (acquired by Cisco in 2004), which pioneered the technology of wide area file systems (WAFS). This technology was adopted by most large enterprises. In 2012, he cofounded Sookasa, Mountain View, CA, USA, a provider of unstructured data security and management for storage cloud services. He is the coauthor of over 170 refereed papers and 27 US patents. His current research involves wire-line and wireless communication networks and on-chip interconnects networks.

Dr. Cidon received the IBM Outstanding Innovation Award twice.

Osnat (Ossi) Mokryn received the B.Sc. degree in computer science and M.Sc. degree in electrical engineering from the Technion—Israel Institute of Technology, Haifa, Israel, in 1993 and 1998, respectively, and the Ph.D. degree in computer science from the Hebrew University, Jerusalem, Israel, in 2004.

She has an extensive high-tech experience from leading companies such as Intel and IBM research labs in Haifa, Israel. She was a Post-Doctorate Fellow with the Electrical Engineering Department, Tel-Aviv University, Tel-Aviv, Israel. Starting in 2008, she was faculty and heading the Internet and Computer Networks field with the School of Computer Science, Tel Aviv Jaffa Academic College, Tel Aviv, Israel. Her recent research focuses on content-aware caching, wireless and opportunistic networks, and data mining for recommender systems.