國立清華大學電機資訊學院資訊工程研究所
碩士論文
Department of Computer Science
College of Electrical Engineering and Computer Science
National Tsing Hua University
Master Thesis

在有限資源限制下的非對稱式冗餘消除演算法
A Resource-Constrained Asymmetric Redundancy Elimination Algorithm

李育賢
Yu-Sian Li

指導教授：徐正炘 博士
Advisor: Cheng-Hsin Hsu, Ph.D.

中華民國 102 年 06 月
June, 2013

中文摘要

本論文針對高速通訊網路下非對稱的頻寬和資源問題，提出了一個在有限資源下的非對稱冗餘消除演算法 (RCARE) ，利用多餘的下載頻寬和接收端的資源，以加速上傳的數據傳輸。該系統可以部署於客戶端或代理伺服器上。RCARE 與現有的非對稱演算法不同，它使用更加靈活的匹配機制來識別冗餘資料，並使用一個傳送端的暫存器吸收過高的下載流量。和現有的冗餘消除演算法相比，它提供了一個可根據資源與效能調整的傳送端暫存器。我們從多個伺服器和校園網路記錄了真實的流量資料，並利用這些資料評估 RCARE 的效能。由我們的模擬結果顯示， RCARE 可比目前的非對稱式通訊演算法達到更高的上傳增益，以及更低的下載流量。我們也為有限資源的傳送端設計了動態調整演算法。此演算法可根據目前的樣本資料，預測並分配資源給目前的數據流，以達到最大的上傳增益。與平均分配資源的基準演算法相比較，動態調整演算法提高了高達 87% 的上傳增益。在前 10% 的實驗結果中 (以最佳的上傳增益排序)， RCARE 平均達到了高達 40.5% 的上傳增益。

Abstract

We focus on the problem of efficient communications over access networks with asymmetric bandwidth and capability. We propose a resource-constrained asymmetric redundancy elimination algorithm (RCARE) to leverage downlink bandwidth and receiver capability to accelerate the uplink data transfer. RCARE can be deployed on a client or a proxy. Different from existing asymmetric algorithms, RCARE uses flexible matching mechanism to identify redundant data, and allocates a small sender cache to absorb the high downlink traffic overhead. Compared to redundancy elimination algorithms, RCARE provides a scalable sender cache which is adaptive based on resource and performance. We evaluate RCARE with real traffic traces collected from multiple servers and a campus gateway. The trace-driven simulation results indicate that RCARE achieves higher goodput gains and reduces downlink traffic compared to existing asymmetric communication algorithms. We design an adaptation algorithm for resource-constrained senders sending multiple data streams. Our algorithm takes samples from data streams and predicts how to invest cache size on individual data streams to achieve maximal uplink goodput gain. The adaptation algorithm improves the goodput gain by up to 87% compared to the baseline. In first 10% of data streams (sorted by the optimal goodput gains), RCARE achieves up to 40.5% goodput gain on average.

Contents

List of Figures

List of Tables

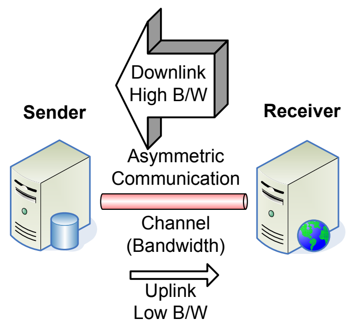
Chapter 1

Introduction

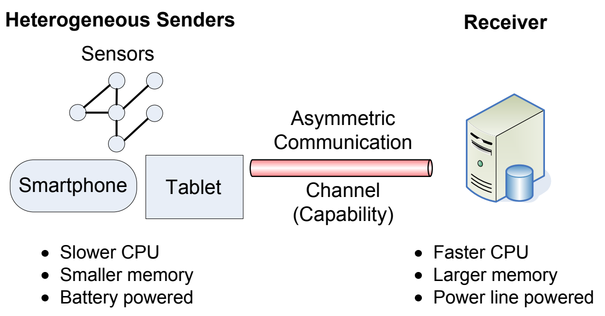
1.1 Overview

Content redundancy often exists in network traffic. Various *redundancy elimination* algorithms [4, 6, 16, 18, 24, 29, 33] leverage the redundancy by sending condensed information rather than redundant data, in order to increase the *goodput*, which is the effective application-level throughput, excluding protocol and error-recovery overhead. We define *goodput gain* of a redundant elimination algorithm as the relative goodput improvement compared to the standard TCP transfer. When the resources at the sender and receiver are unconstrained, the existing redundancy elimination algorithms may achieve nontrivial goodput gains. However, many network communication scenarios have *asymmetric* resource constraints in terms of network *bandwidth* and end-device *capability*.

Fig. 1.1(a) illustrates the bandwidth asymmetry. This scenario often occurs in the access networks. Example access networks with asymmetric bandwidth are Asymmetric Digital Subscriber Lines (ADSLs), cable modems, 3G/4G cellular networks, and hybrid satellite-terrestrial access [7, 23], in which a high speed satellite downlink is paired with a telephone line for uplink traffic. The downlink bandwidth of these channels could be up to 1,000 times higher than the uplink bandwidth [9, 12, 31], due to business concerns and technology limitations. Sharing large files, such as documents, music, videos, and pictures over these channels results in long upload time, sluggish playouts, and degraded user experience. Fig. 1.1(b) shows capability asymmetry among hosts. In this scenario, end-devices, such as smartphones and sensors, have limited memory size, processing power, and battery capacity. Moreover, the end-devices are often heterogeneous in capability. Although these hosts are not capable to run complex algorithms, they are often connected to powerful servers and clouds. Recently, the negative impacts of capability asymmetry are gradually surfacing, for example: (i) more smartphone applications push computations into the cloud, which may dramatically increase the network traffic [21], and (ii)



(a)



(b)

Figure 1.1: We consider both: (a) bandwidth and (b) capability asymmetric scenarios.

the Internet of Things (IoT) paradigm connects a huge number of sensors to the public Internet, which imposes tremendous traffic load [8, 36].

We collectively call network communications over bandwidth and capability asymmetric channels as *asymmetric communications*. The existing redundancy elimination algorithms [4, 6, 16, 18, 24, 29, 33] can not: (i) utilize excessive downlink bandwidth to increase uplink goodput gain in bandwidth asymmetric scenarios, nor (ii) be executed on resource-constrained end-devices in capability asymmetric scenarios. Therefore, users of asymmetric communications have to resort to upgrading their channels or end-devices, which are costly and could render some business models less viable. Hence, a redundancy elimination algorithm designed for resource-constrained asymmetric communication is desirable.

In this thesis, we study the problem of increasing uplink goodput in asymmetric communications by capitalizing on the otherwise wasted downlink bandwidth and receiver capability. More specifically, we design a new asymmetric communication algorithm on top of the transport protocols to maximize the uplink goodput gain. Several asymmetric communication algorithms have been proposed in the literature [3, 10, 13, 14, 22, 32]. Trang et al. [28] use synthetic traces to evaluate the performance of these algorithms, but their performance on actual network traffic has never been studied. In the thesis, we employ real network traces to evaluate the existing asymmetric communication algorithms, and identify their limitations.

To cope with their limitations, we develop a *parameterized* Resource-Constrained Asymmetric Redundancy Elimination (RCARE) algorithm, which is general in the sense that many system parameters can be adjusted on-the-fly. We also propose an adaptation mechanism to optimally allocate the resources among data streams. We empirically study the data stream *characteristics*, *quantified by several features*, such as entropy and packet size. In particular, several data stream features are computed from the first few hundreds of network packets belonging to a data stream, and the optimal allocation arrangement is

Table 1.1: Per-packet performance of dynamic algorithms

Algorithm	Bits from Receiver	Bits from Sender	Rounds
DBES	$(H + O(1)) \log N$	$H + \frac{n \log N}{M} + O(1)$	$H + O(1)$
TreeQuery	$2N - 1$	$H + \frac{n \log N}{M} + O(1)$	1
ListQuery	$\lfloor N^{1/k} \rfloor \lceil \log N \rceil$	$kH + \frac{n \log N}{M} + O(1)$	1
QueueQuery	$\lfloor N^{1/k} \rfloor \lceil \log N \rceil$	$kH + \frac{n \log N}{M} + O(1)$	1

decided based on an empirically-trained *model* so as to maximize the goodput gain. The model is derived using a large set of real life network traces collected from a high-speed campus network [20]. Trace-driven simulations indicate that the proposed RCARE algorithm outperforms the existing asymmetric communication algorithms and is close to redundancy elimination algorithms. Moreover, RCARE is much more flexible and suitable under heterogeneous network and host resource constraints.

The rest of this thesis is organized as follows. In this chapter, Sec. 1.2 surveys the related work in the literature. We conduct trace-driven simulations in Sec. 1.3 to quantify the performance of the existing asymmetric communication algorithms. Based on the findings made in Sec. 1.3, we propose a new asymmetric algorithm and evaluate the performance in Ch. 2. Ch. 3 proposes an adaptation mechanism for multiple data streams and shows the goodput gains of RCARE. Ch. 4 concludes this thesis.

1.2 Related Work

1.2.1 Asymmetric Communication Algorithms

The existing asymmetric communication algorithms can be categorized into two classes: static and dynamic. The static asymmetric communication problem is first considered by Adler and Maggs [3]. The problem is static because it assumes the communication packets follow a known and fixed probability distribution. In particular, the problem considers a sender sending M packets to a receiver. The packets are chosen from N possible packets following a probability distribution that is known to the receiver but not the sender. The entropy of this distribution is denoted as H. Adler and Maggs propose two algorithms to increase the uplink goodput. The static problem is also considered in other work [10, 22, 32], in which new algorithms are proposed. We refer to these algorithms as static algorithms.

It is reported that the static problem is built on top of a strong assumption that the receiver knows the probability distribution of packets, which is unrealistic [13, 14]. Gagie therefore considers a dynamic problem, in which multiple clients send M packets to a server, and these packets are chosen from n packets (out of N possible packets in total) following a probability distribution that is unknown to both senders and the receiver [13, 14]. Gagie writes the distribution entropy as H, and proposes algorithms for the dynamic problem [13]. We refer to these algorithms as dynamic algorithms. In our work, we do not assume a receiver knows the probability distribution of the packets, and thus we only consider dynamic algorithms throughout this thesis.

We present the main idea behind dynamic algorithms below. Generally, each asymmetric communication algorithm maintains a *cache* at the receiver, to keep track of t *seen* packets sent from one or more senders, where t is a system parameter. The receiver uses this cache in the following way. For each incoming packet, the receiver *guesses* the packet according to the cache, and *asks* the sender if the guess is correct. The sender either: (i) confirms the correctness of the guess and moves on to the next packet or (ii) sends the receiver a *hint* to adjust its guess on the same packet in the next round. Each packet is delivered in multiple rounds $r \geq 1$. A receiver updates its cache once successfully receiving a packet, in order to leverage on the known packet pattern for fewer guess rounds.

We emphasize that the cache is online and consists of not only seen packets but also their statistics such as hit counts and last-seen timestamps. Since the packet distribution is unknown and dynamic, the receiver may saturate all educational guesses, and has to ask the sender to transmit the packet as-is. A sender also sends a packet as-is if the number of rounds r exceeds an algorithm-specific threshold $r_{\max}$, in order to avoid long latency.

Table 1.1 summarizes the asymptotic performance of various asymmetric communication algorithms. We briefly describe the algorithms below.

- **Dynamic Bit-Efficient-Split (DBES)** uses a leaf oriented binary search tree of seen packets as the cache, in which distinct packets are stored as lexicographically sorted leaf nodes. To make a guess on each new packet, the receiver traverses the cache from its root, and transmits the packet stored at the current node to the sender. The sender replies with *smaller, larger,* or *same*. With smaller/larger, the receiver then descends one-level closer to the leaf nodes. When running out of nodes, the receiver requests the sender to send the packet itself. DBES sets $r_{\max} = \infty$.
- **TreeQuery**, different from DBES, it reduces the number of communication rounds by encoding the whole binary search tree into a *data bundle*. The receiver transmits the binary tree to the sender, and the sender replies with the traversal path also in a single data bundle. The sender transmits a packet as-is if it cannot be found in the tree. TreeQuery sets $r_{\max} = 1$.
- **ListQuery** maintains a cache of seen packets sorted by their hit counts in non-decreasing order. Different from TreeQuery, ListQuery uses a parameter $k \geq 1$ to control the amount of downlink traffic. That is, the receiver encodes the sublist of

the $t^{1/k}$ most popular packets into a data bundle and transmits it to the sender. If the new packet is in the sublist, the sender transmits its identification; otherwise, the sender transmits that packet itself. ListQuery sets $r_{\max} = 1$.

- **QueueQuery** is very similar to ListQuery. The main difference is, instead of a sublist of most popular packets, the receiver transmits a queue of $t^{1/k}$ most recently seen packets to the sender.

Trang et al. [28] implement DBES and ListQuery algorithms in NS-2 simulator [25]. They conduct simulations with synthetic traces and observe 24% uplink goodput gain. In my thesis, we also conduct trace-driven simulations quantify the potential of DBES, ListQuery, and QueueQuery using real network traces. The results are indicated in Sec. 1.3.

1.2.2 Redundancy Elimination Algorithms

There are several redundancy elimination algorithms proposed in the literature, which can also be used to increase uplink goodput of asymmetric communications to some degree. These algorithms can be categorized into two classes: online compression algorithms [18, 24, 33] and protocol-independent redundancy elimination algorithms [4, 6]. Yang et al. [18, 33] implement LZ77 compression algorithm with VLSI architecture. It uses the content addressable memory to design a high-speed data compressor. Munteanu et al. [24] use LZ compression algorithm to do the packet compression on the fly. They evaluate the performance using HTTP traffic. The results reveal that it can reduce the traffic up to 38%. Online compression algorithms compress the data payload in real time and have been deployed in commercial routers [29]. Protocol-independent redundancy elimination algorithms remove duplicated packets. For example, in Aggarwal et al. [4], the server maintains per-client caches for downlink redundancy elimination. Some of these algorithms have been employed as WAN optimization techniques [1].

In redundancy elimination algorithms, the fingerprinting algorithm is one of the main factors in detecting redundant data. The fingerprinting algorithm and it's configuration also affects the processing speed. Aggarwal et al. [4] evaluate the performance of current fingerprinting algorithms such as MODP, MAXP, and FIXED. They then propose a new fingerprinting algorithm, called SAMPLEBYTE, which is much faster than current algorithms. Halepovic et al. [15] study on the influence of sampling overlap and oversampling. Overlap and oversampling incur much higher overhead but they improve redundancy elimination by 9-14%. They observe the content type also affect the performance. For example, text data has higher probability to achieve better results than compressed data. They propose a method to dynamically adjust the sampling period based on the textiness ratio. With the method, they can avoid unnecessary oversampling to reduce overhead.

In contrast to RCARE, while redundancy elimination algorithms [4,6,16,18,24,29,33] may reduce the uplink traffic amount, they cannot leverage on downlink bandwidth and receiver capability for faster upload speed. Moreover, existing redundancy elimination algorithms demand considerable resources, including memory, CPU cycles, and energy at senders, and thus are not suitable when the senders have limited resources. Last, unlike the proposed RCARE, they cannot leverage redundancy across multiple senders.

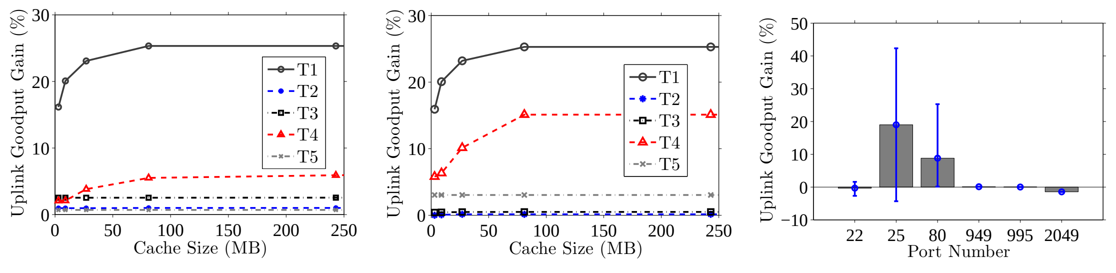
Recently, Zohar et al. [35] study the downlink redundancy elimination problem in clouds. They propose a *receiver-driven* redundancy elimination algorithm, called PACK to shift the resource demands from the senders in clouds to receivers on residential/corporate networks, so as to reduce the cost of cloud customers. PACK links transmitted data together, which is like a chain. It uses these chains to predict the next incoming data so as to reduce the transmitting cost. PACK is probably the work closest to the proposed RCARE algorithm, despite PACK was designed to minimize the cost of cloud customers at the expense of potentially overloading mobile computers and sensors that are downloading from cloud. While PACK shifts all the overhead from a sender to a receiver, our RCARE allows dynamic resource allocations between the sender and receiver for higher goodput gains without overloading the receivers. The authors of [35] mention that PACK is more suitable to larger objects such as video and email attachments, and recommend to fall back to sender-driven (traditional) redundancy elimination algorithm [35] for smaller objects. In fact, even for videos, PACK leads to slightly lower goodput gain compared to a sender-driven redundancy elimination algorithm as reported in [35].

Compared to RCARE, the existing redundancy elimination algorithms [1, 4, 6, 18, 24, 29, 33, 35] suffer from three limitations: (i) they require high computational power at the sender to identify data redundancy and large storage space for caching historical traffic, which are not available at resource-constrained senders, (ii) they do not leverage redundancy among data from different senders, which could be nontrivial, especially in mobile and IoT applications, and (iii) they do not capitalize the already-paid downlink bandwidth and receiver capability.

Last, a preliminary version of this work was presented in Li et al. [19]. This thesis contains a new adaptation algorithm, more extensive analysis, and additional simulation results, compared to [19].

Table 1.2: Packet traces from real servers

Trace	Server Type	Location	Duration (hr)	Size (MB)
T1	Enterprise Server	US East Coast	168	59
T2	Enterprise Server	US West Coast	98	153
T3	Home Server	Taiwan	60	404
T4	Home Server	US West Coast	122	821
T5	University Server	Canada West Coast	47	12,568



(a) (b) (c)

Figure 1.2: Goodput gains achieved by ListQuery, results from: (a) protocol-independent cache, (b) HTTP cache, and (c) per-protocol caches with cache size 1 GB.

1.3 Limitations of Current Asymmetric Communication Algorithms

Table 1.1 presents the asymptotic performance of the current asymmetric algorithms. While the asymptotic analysis sheds some lights on the effectiveness and efficiency of the algorithms, it does not reveal the potential of these algorithms in real life scenarios. In this section, we collect actual network traces from real servers, and conduct trace-driven simulations to quantify the performance of these algorithms.

1.3.1 Potential of Asymmetric Communication Algorithms

We collected egress packet traces from five real servers in enterprise, home, and university networks using `tcpdump`. All the servers ran Linux, and had 4-12 local users. We collected the traces without asking users to change their daily usage patterns. Some services, e.g., Web services, may have many anonymous remote users. Table 1.2 summrized the information of individual traces. The trace files enable us to perform realistic trace-driven simulations.

Trang et al. [28] implemented the DBES, ListQuery, and QueueQuery algorithms in the NS-2 simulator. We however found that conducting NS-2 simulations is quite time consuming. Therefore, we also developed our own event-driven simulator using C/C++, which runs more than 100 times faster than NS-2 when the network topology is simple.

We ran several simulations using both NS-2 and our simulator, and carefully compared the simulation results to verify the correctness of our simulator. In the rest of this work, we report the simulation results from our own simulator.

We let the packet size be 1,500 bytes, and vary the cache size from 3 to 1,500 MB. We set $k = 1$ for ListQuery and QueueQuery algorithms. We conduct two sets of simulations. First, we use each trace file to drive the simulator with one of the three considered algorithms. This is to emulate the scenarios where a protocol-independent cache is used between any pair of sender and receiver. Second, we split each trace file into smaller trace files based on their port numbers. We then run the simulators with the split trace files, so as to emulate the scenarios where a per-protocol cache is employed. We use the uplink goodput gain as the performance metric. The goodput gain is defined as the relative goodput increase of an asymmetric communication algorithm compared to a standard TCP data stream.

The simulation results indicate that DBES never results in positive goodput gain. Furthermore, throughout our simulations, QueueQuery always achieves similar, but slightly worse uplink goodput gain compared to ListQuery. Therefore, we only report results from ListQuery. We found that the uplink goodput gain does not increase when cache size is larger than 250 MB; hence, we only plot the results with cache size in $[3, 250]$. We first present the results from ListQuery. We plot its protocol-independent uplink goodput gain in Fig. 1.2(a). This figure shows that only one trace (T1) results in uplink goodput gain higher than 7%; three traces (T2, T3, and T5) lead to negligible ($< 2\%$) uplink goodput gain. Next, we report the per-protocol uplink goodput of HTTP traffic in Fig. 1.2(b). Compared to Fig. 1.2(a), the uplink goodput gain of HTTP is generally higher. Nevertheless, majority of the traces (T2, T3, and T5) still lead to small ($< 3\%$) uplink goodput gain. Last, we compute the uplink per-protocol goodput gain of individual traces, and plot their mean, minimum, and maximum gains in Fig. 1.2(c). This figure shows that the uplink goodput gains are low, with exceptions of the HTTP (port 80) and SMTP (port 25) protocols. Even for HTTP and SMTP, the worst per-trace uplink goodput gains are $< 10\%$.

1.3.2 Discussion

We take a closer look at the packets in the traces to determine the root causes of the inferior performance of the current asymmetric communication algorithms. We found that these algorithms are limited in the sense that they only leverage the redundancy of *exact-match* packets. In actual traffic traces, however, exact-match packets do not occur too often. Rather, we often observe packets that are almost matching except a few *critical bytes* that are different from one another. Despite there is a high redundancy between the two packets, the current algorithms will treat them as different packets. Furthermore, a common byte range may appear in different positions of two packets, which are then considered as different packets by the current algorithms. Take HTTP packets as examples, meta-data such as timestamps, cookie IDs, and hit counts are critical bytes, which may have variable length. This in turn results in diverse *offsets*. We refer to packets that only differ by critical bytes and diverse offsets as *partial-match* packets. We believe that the current algorithms achieve low uplink goodput gains because they cannot identify the partial-matches. We develop a new asymmetric communication algorithm to address this limitation in the next section.

1.4 Contributions

This thesis makes the following main contributions:

- We propose an asymmetric redundancy elimination algorithm, RCARE, which, to the best of our knowledge, is the first redundancy elimination algorithm tailored for resource-constrained asymmetric communications.

- We study the correlation between unlink goodput gain and data stream features, and derive an adaptation algorithms for allocating the cache size based on data stream features.

- Extensive trace-driven simulations show that RCARE outperforms the state-of-the-art asymmetric communication algorithms [13, 14] by far: up to 50 times improvement on uplink goodput gain and up to 384 times reduction on downlink traffic amount are observed.

Chapter 2

RCARE

In this chapter, we present a first efficient asymmetric redundancy elimination algorithm. We conduct trace-drvien simulations to evaluate the performance and compare it with existing algorithms.

2.1 A new Asymmetric Communication Algorithm: RCARE

We present the RCARE algorithm in this section. For the ease of presentation, we consider a single data stream in this section. Other deployment scenarios will be discussed in Ch. 3.

2.1.1 Overview

The main objective of RCARE is to maximize the uplink goodput gain by supporting partial-match, which allows us to capitalize common byte ranges with arbitrary offsets and lengths shared between the current and a historical packet. The secondary objective of RCARE is to be parameterized, in order to adapt to data streams with diverse characteristics. RCARE resides in between the transport and application layers, and provides a boosted uplink data transfer service to applications. Fig. 2.1 illustrates that RCARE can be deployed on two hosts of asymmetric communications (Fig. 2.1(a)), or on an in-network proxy (Fig. 2.1(b)). Fig. 2.1 only presents a simplified network topology. More elaborated topologies are possible. For example, two hosts of asymmetric communications may connect through a common proxy for goodput gains in *both* directions. Multiple proxies at different Internet Service Providers (ISPs) may also collaborate with each other by establishing high-bandwidth channels among them.

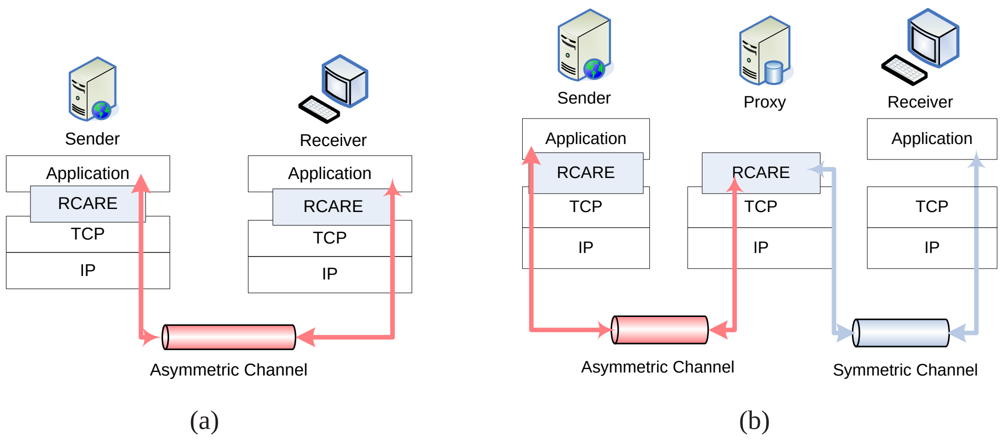


Figure 2.1: RCARE can be deployed on: (a) hosts and (b) proxies.

2.1.2 Packet Caches

Similar to existing asymmetric communication algorithms, RCARE maintains a cache of historical packets at the receiver. We let the receiver cache size be B_r MB, which is determined by the receiver's capability. RCARE also allocates a packet cache at the sender, and we let the sender cache size be B_s MB, which is determined by the sender capability. At the sender, each new packet is compared against the historical packets in the sender cache to find the longest common byte range. The matching byte ranges are then encoded for reducing the uplink data redundancy.

In RCARE, the receiver cache cannot be smaller than the sender cache, otherwise some encoded byte ranges might not be decodable at the receiver. Therefore, we have $B_r \geq B_s$. Having a larger receiver cache makes sense for capability-constrained mobile and sensing devices, because the receiver can *help* the senders to memorize more historical packets for higher uplink goodput gain. More specifically, the receiver periodically transmits a subset of the receiver cache to the sender. The sender then uses this cache subset to replace the old, potentially outdated, sender cache. This is referred to as *cache update*. The receiver also keeps a copy of the sender cache for the decoding purpose, which is called sender cache *shadow* in RCARE.

The cache update is performed once every f packets, where *update frequency* f is a system parameter. The update frequency controls the tradeoff between downlink traffic amount and uplink goodput gain, because less frequent updates result in more outdated sender cache, but save some downlink traffic. In RCARE, we assume the size of each cache update is B_s for simplicity. That is, the receiver always *fills up* the entire sender cache in each cache update. [1]

Given that $B_r \geq B_s$, RCARE may not copy the entire receiver cache to the sender. Therefore, RCARE has to define a *selection policy* to maximize the chance of identifying

[1]More elaborated update strategies are possible, e.g., partial cache updates or incremental cache updates

common byte ranges at the sender for higher uplink goodput gain. Typical selection policies include: (i) Most-Recently-Used (MRU) and (ii) Most-Frequently-Used (MFU) packets. We consider a general *hybrid* policy P_β ($0 \leq \beta \leq 1$), which selects β MRU and $1 - \beta$ MFU packets. It is clear that P_β covers the full spectrum of selection policies between (and including) MRU and MFU. Upon there is a common byte range falls in a packet, RCARE increases its hit count by one and/or updates its last-seen timestamp. Different from f, β does not affect the downlink traffic amount, yet may affect the uplink goodput gain.

2.1.3 Efficient Partial-Match Algorithm

A simple approach to find common byte ranges between the current packet and a single historical packet in the sender cache is to traverse through *every* single byte of that historical packet. Then the simple approach checks all historical packets in the sender cache. Such a naive approach is clearly not feasible in real-time systems given the huge number of comparisons to be done. To speed up the partial-match process, we employ the following techniques: (i) selecting representative windows, (ii) hashing representative windows, (iii) locating matching byte range, and (iv) encoding the matching byte range. These techniques are inspired by redundancy elimination algorithms in the literature [4, 5, 15, 27]. We discuss these techniques in details below.

Selecting representative windows. To avoid excessive computational complexity at the receiver, each packet is scanned and marked with one or multiple *representative* windows, where each window is w-bytes long. We refer to w as the window size. The partial-match process uses these representative windows as *entering* points to locate matching byte ranges and thus the complexity can be controlled. Moreover, we use a window sampling frequency p to throttle the number of representative windows. In particular, RCARE only considers $1/p$ qualified representative windows for the sake of lower computational complexity. w and p are system parameters, and could affect the performance of RCARE. We empirically compared several w and p values and found that $w = 32$ and $p = 64$ result in a good tradeoff between running time and uplink goodput gain.

After determining the window size and sampling frequency, we need to design a policy on choosing the representative windows. Aggarwal et al. [4] propose a policy called SAMPLEBYTE, and show it outperforms other policies. We adopt SAMPLEBYTE in RCARE. Specifically, the receiver maintains a *marker* list of m byte values, where $1 \leq m \leq 256$. Whenever the receiver sees a new packet, it traverses through every byte of that packet, and compares its value against the markers' values. If there is a match at offset x, the receiver selects $[x, x + w - 1]$ as the representative window, and skips $p/2$ bytes in order to comply with the sampling frequency. RCARE dynamically computes the marker list based on the occurrence frequency of all byte values across the receiver cache. This is closer to a dynamic approach recently proposed by Halepovic et al. [15]. Different from redundancy elimination algorithms [4, 5, 15, 27], RCARE pushes the complexity of computing the marker list, along with other computations, to the powerful receiver. RCARE employs a marker list refresh threshold T_m packets, for statistically meaningful marker lists. The receiver updates the marker list once every $r_m = \max(f, T_m)$ packets, and transmits the list to the sender. We let $T_m = 1,000$ if not otherwise specified.

Hashing representative windows. To facilitate fast lookup, we employ Jenkins Hash function [17] to compute a 32-bit hash code, referred to as *fingerprint*. The receiver maintains a hash table with fingerprint as keys, and <historical packet ID, offset> as values, where historical packet ID points to a specific packet in the cache. This is called the fingerprint table, which is sent to the sender whenever the receiver does a cache update. The sender uses this fingerprint table for common byte range lookups.

Locating matching byte range. For each packet, the sender uses the marker list to locate all representative windows in it. The sender then computes their fingerprints. Comparing against the fingerprint table, the sender finds the first matching window. It then expands the matching window to the left and right one byte after another, so as to maximize the matching byte range. This is similar to the strategy proposed in [4, 5, 27].

Encoding the matching byte range. The sender sends <historical packet ID, offset, length> instead of the byte range itself. The receiver uses the sender cache shadow to reconstruct the original byte range. Given that the tuple is much shorter than the raw byte range, RCARE may achieve high goodput gain.

2.2 Trace-Driven Simulations

We quantify the performance of RCARE in this section.

2.2.1 Setup

We extend the event-driven simulator presented in Sec. 1.3.1 to support RCARE. We compare RCARE against ListQuery, because ListQuery outperforms all other asymmetric communication algorithms in terms of the uplink goodput gain, which is also shown in Sec. 1.3.1. We also implement EndRE [4] and GZip [34] algorithm in the simulator for comparisons. The EndRE algorithm employs symmetric caches for sender-driven redundancy elimination, while GZip compresses the payload of each packet before sending it out. We use actual network traces collected in Sec. 1.3.1 (see Table 1.2) to drive the simulator. We conduct both protocol-independent and per-protocol simulations. In the

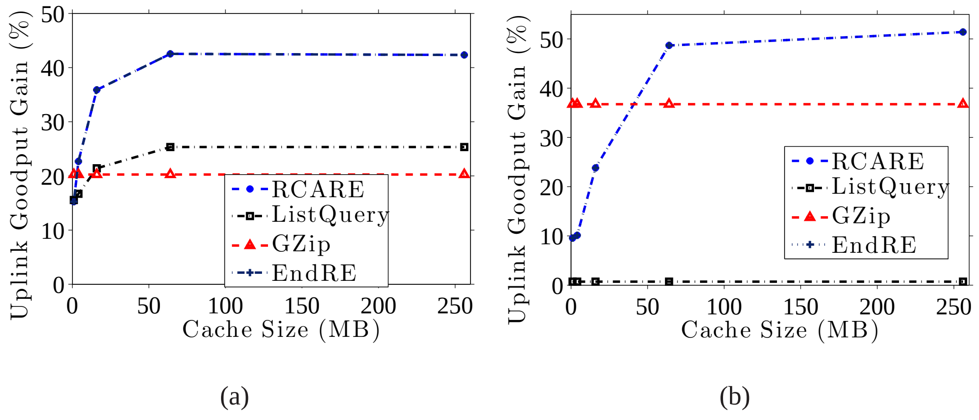


Figure 2.2: Uplink goodput gain achieved by various protocols, sample results from: (a) T1 and (b) T5.

latter case, we split each network trace into multiple protocol-specific traces. We ignore the protocols with fewer than 1,000 packets. We program the simulator to report the performance results after each round of simulation. We consider the following performance metrics: (i) uplink goodput gain in percentage, (ii) relative overhead, which is defined as the ratio of downlink traffic amount and the raw uplink traffic amount in percentage, and (iii) per-packet encoding and decoding time in msec.

For RCARE and EndRE, we let marker list length $m = 10$, update frequency $f = 1,000$, receiver cache size $B_r = 64$, sender cache size $B_s = 16$, selection policy parameter $\beta = 0.1$, and trace file be T1, if not otherwise specified. We emphasize that, for fair comparisons, the sender and receiver cache sizes *include* all the storage overhead, in particular the fingerprint tables. For EndRE, the sender and receiver cache sizes must be identical, while the proposed RCARE allows the users to specify a smaller sender cache size. Various system parameters, including the update frequency, selection policy parameter, and sender cache size, are varied in the simulations to study their implications on system performance.

2.2.2 Results

Improved uplink goodput gain. We first compare the uplink goodput gain achieved by all considered algorithms. For fair comparisons, we let $B_r = B_s \in \{1, 4, 16, 64, 256, 512\}$ and $f = 1$ since EndRE only supports this configuration. We plot the sample results from T1 and T5 in Fig. 2.2, in which we skip $B_r = B_s = 512$ for brevity, as it leads to the similar results as $B_r = B_s = 256$. Note that, in this figure, RCARE achieves similar uplink goodput gain as EndRE; therefore their lines overlap with each other. Fig. 2.2(a) shows that RCARE outperforms GZip when $B_s = B_r \geq 4$. Moreover, RCARE always outperforms ListQuery: up to 1.54 times of uplink goodput gain is possible. Fig. 2.2(b)

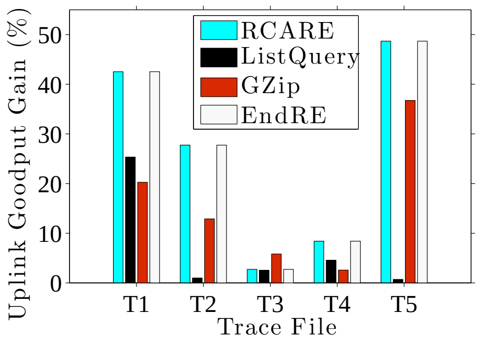


Figure 2.3: Uplink goodput gain achieved by various protocols, overall results from all traces.

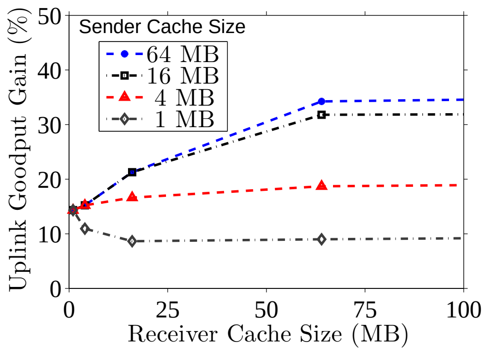


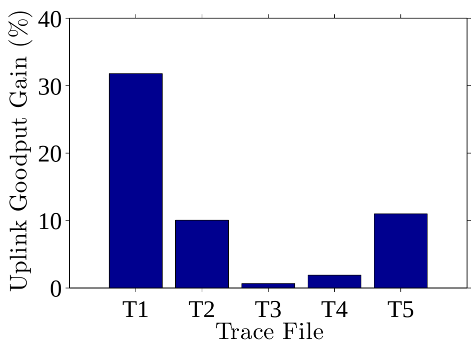
Figure 2.4: Implications of B_r and B_s on uplink goodput gain.

shows that RCARE significantly outperforms ListQuery: about 50 times uplink goodput gain improvement is observed. Fig. 2.3 present the overall results. This figure shows that RCARE constantly outperforms ListQuery. Given that ListQuery and GZip lead to inferior performance, and EndRE dictates $B_s = B_r$ (thus is inflexible), we concentrate on the evaluation of RCARE in the rest of this section.

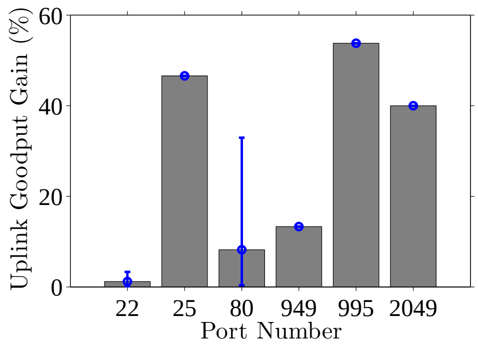
Implications of B_r and B_s. We vary $B_r \in \{1, 4, 16, 64, 256, 512\}$, and $B_s = \{1, 4, 16, 64\}$. We plot the uplink goodput gain in Fig. 2.4, in which we zoom into $B_r \in [0, 100]$. We make two observations. First, with $B_s \geq 4$, larger receiver cache leads to higher uplink goodput gain. Second, when sender cache size $B_s = 1$, larger receiver cache actually leads to *lower* goodput gain. We believe this is because larger B_r means more room for selecting representative windows, and thus scenarios with small B_s are more sensitive to the quality of window selection policy. Fig. 2.4 reveals the tight correlation between B_s and B_r: a joint decision on them need to be made for a good tradeoff between uplink goodput gain and resource consumption. The algorithm to dynamically adjust B_s and B_r are shown in Ch. 3.

Diversity of uplink goodput gain. We plot the protocol-independent and per-protocol uplink goodput gain in Fig. 2.5. This figure shows that the achieved gain of RCARE is quite diverse: 3–32% for protocol-independent case (Fig. 2.5(a)) and 2–57% for per-protocol case (Fig. 2.5(b)). In particular, SMTP (25), POP3S (995), and NFS (2049) achieve more than 40% gains. Fig. 2.5(b) also shows that the range of gains of a protocal with different traces could be large. For example, the HTTP protocol with different traces achieves very different gains. Hence, the uplink goodput gain of RCARE highly depends on the payload content.

Tradeoff between uplink goodput gain and relative overhead. While larger sender

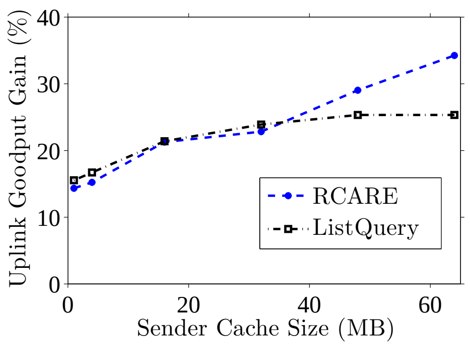


(a)

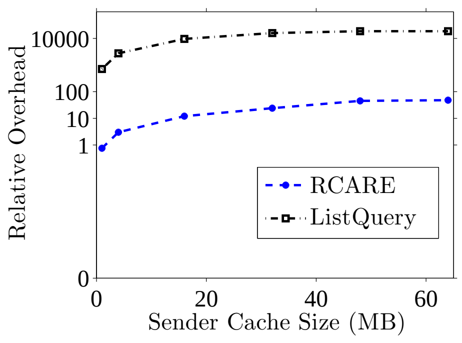


(b)

Figure 2.5: Diverse goodput gains from: (a) various trace files and (b) per-protocol trace files.



(a)



(b)

Figure 2.6: Tradeoff between: (a) uplink goodput gain and (b) relative overhead.

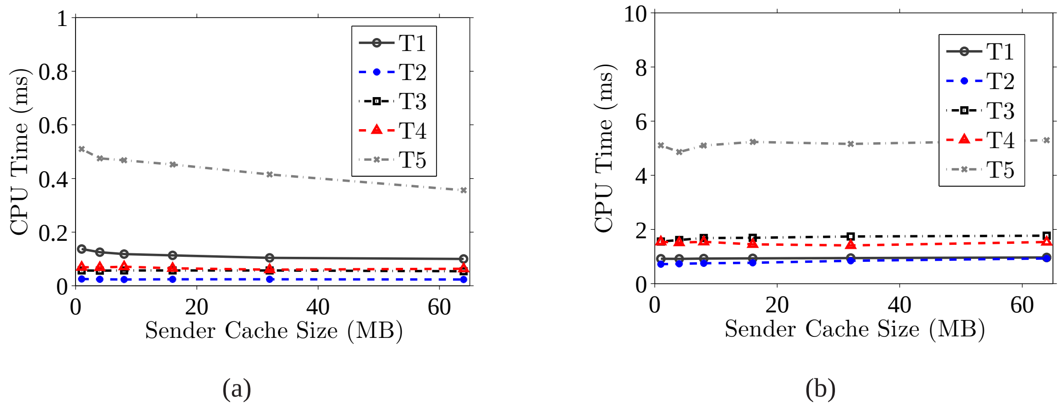


Figure 2.7: Time complexity of RCARE, per-packet: (a) encoding and (b) decoding time.

cache results in higher uplink goodput gain, it also leads to more downlink traffic. We plot the uplink goodput gain and relative overhead of RCARE and ListQuery with $B_s \in \{1, 4, 16, 32, 48, 64\}$ in Fig. 2.6. Fig. 2.6(a) shows that RCARE and ListQuery lead to similar gain when $B_s \leq 32$, and RCARE outperforms ListQuery when $B_s > 32$. Fig. 2.6(b) shows the relative overhead. It clearly illustrates that ListQuery suffers from huge relative overhead: it incurs up to 18,450 times of downlink traffic amount, compared to the raw uplink traffic amount. In contrast, RCARE uses a small sender cache to *absorb* a huge portion of the downlink traffic, and only incurs at most 48 times of downlink traffic amount (not visible in this figure due to the Y-axis scale), which is almost 384 times lower than that of ListQuery.

Encoding and decoding time. Fig. 2.7 shows per-packet encoding and decoding time, collected from a commodity 3.4 GHz Intel i7 Linux PC. We set receiver cache size to $B_r = 64$ and vary sender cache size B_s. This figure demonstrates the efficiency of RCARE algorithm: encoding overhead is less than 0.5 msec while the decoding overhead is less than 5.2 msec. Moreover, RCARE is scalable to the cache size because the lines in this figure are almost flat. We take a closer look and find that the decoding time in Fig. 2.7(b) is increasing with a small slope. This is because of receiver needs more resources to compute the cache update for the server when B_r increases. In these two figures, the T5 costs more CPU times because it has longer payload size than others. For example, the average payload size of T5 is about 2,000 bytes, while that of T2 is only 300 bytes. We acknowledge that despite we consider all the computational overhead in the simulator when calculating encoding/decoding time, the reported results might not include certain overhead in real systems, e.g., the overhead incurred by the network stack and the multitasking overhead of Operating Systems. Nonetheless, we believe our RCARE can run in real time after some code optimization. Integrating RCARE in a real network stack is our future work.

2.2.3 Observations

We made two observations on the performance of RCARE algorithm out of the simulation results. First, RCARE outperforms the existing asymmetric communication algorithms by a large margin. Second, the achieved goodput gain varies a lot due to: (i) diverse characteristics of data streams and (ii) different system parameters. In extreme cases, RCARE may lead to negligible goodput gain, or even result in goodput *loss*. Therefore, how to select the best system parameters for each data stream is critical to the performance of RCARE. We conduct extensive simulations in the next section, and derive an adaptation mechanism for this purpose.

Chapter 3

Dynamic Adaptation Algorithm and Large-Scale Simulations

In this chapter, we conduct simulations to study how RCARE's system parameters and data stream characteristics affect the uplink goodput gain. We then propose a practical adaptation algorithm for RCARE.

3.1 Data Stream Analysis

In order to study how system parameters and data stream characteristics affect the uplink goodput gain, we collect real-life traffic trace for further analysis. We then recommend the best configuration for later simulations.

3.1.1 Real-Life Traffic Trace Collection

We need a larger set of packet traces to make statistically meaningful analysis. To achieve that, we collected actual packet traces from a campus network [20] at National Chiao Tung University, Taiwan. More specifically, we ran `tcpdump` on an access router to record all the packets, including the headers and payloads. With a 2 TB hard disk, we collected 10-hour traces from 12:00 to 22:00 on February 20th, 2012. Fig. 3.1 shows the per-hour traffic amount, which reveals that 7 p.m. is the peak hour. We recorded 1,632 GB packet data in total. The traces contain 3,358 distinct IPs on the local network and 3,598,829 distinct IPs from the Internet. We divided the traces into data streams, based on each packet's 4-tuple, including source/destination IP/port. To ensure that each data stream has at least a chance to update the cache, and to avoid high setup overhead of short-living data streams, we only consider data streams longer than 2 MB data amount. This gives us 22,330 data streams for our analysis presented below.

3.1.2 Data Stream Characteristics

The data streams have diverse characteristics, which may be described by traffic *features*. The traffic features affect the performance of the RCARE algorithm, and provide hints on selecting the best system parameters for optimal uplink goodput gain. We consider the following data stream features in this article.

- *Port number*. Different protocols may inherently carry different amount of redundancy. For example, HTTP protocol mostly carries unencrypted data, which is easier to compress, while HTTPS protocol carries encrypted data, which is harder to compress. The port number is a good hint for the employed network protocol of a data stream. We consider the source port throughout this thesis.

- *ASCII ratio* θ. ASCII data generally contains more redundancy, compared to binary data. Therefore, the percentage of data in ASCII may be a good indicator for data redundancy.

- *Entropy* H. The average entropy of the identified byte ranges is the most direct indicator of the expected information amount carried by the byte ranges. However, computing average entropy of *variable-length* byte ranges is complex, and thus we compute average entropy of 32-byte long *fixed-length* data blocks. We empirically found that the entropy of 32-byte blocks approximates the entropy of variable-length byte ranges well, compared to 4-, 8-, 16-, and 64-byte data blocks.

- *Mean packet length*. Large packet length might lead to lower header and control overhead and thus higher goodput gain.

- *Standard deviation of packet length*. More uniform packet length may indicate bulky data transfers, which may consist of more data redundancy.

Depending on the features of each data stream, a different set of system parameters may be selected for maximizing the goodput gain. The proposed RCARE algorithm takes the following system parameters: Receiver cache size B_r, Sender cache size B_s, Marker list length m, Selection policy parameter β, and Update frequency f. We study how the data stream features and system parameters affect the goodput gain in the next section.

To simplify the problem of choosing the best parameters, we define multiple profiles for data streams with different features. Each profile consists of a set of pre-defined parameters, which will be empirically derived in the rest of this section. In particular, this is done by conducting extensive trace-driven simulations with diverse traffic traces.

3.1.3 Simulation-Based Analysis

We modify the simulator used in Sec. 2.2.1, so that each data stream is considered separately. In particular, we create a separate cache for each data stream, and apply RCARE with different parameters to each data stream. We record the resulting goodput gain for each set of parameters. We also analyze and report the features of each data stream. We note that although we consider all the data blocks when analyzing the data stream features, only the first α data blocks of each new data stream are sampled in real systems. That is, we will use sample features to approximate the features of the complete data streams.

We choose three-hour sample traces from 12:00, 15:00, and 19:00. We use each data stream to drive our RCARE simulator and calculate the average gain of all the 8,147 data streams from the sample traces. We consider three simulation scenarios, in which we assume $B_r = B_s$ for simplicity. If not otherwise specified, we let $B_r = B_s = 4$, $m = 10$, $\alpha = 1,000,000$, $\beta = 0.1$, and $f = 1,000$. In scenario I, we vary $m = \{5, 10, 20, 40, 80\}$. In scenario II, we let $\beta = \{0, 0.1, 0.25, 0.5, 0.75, 1\}$. In scenario III, we choose $f = \{10, 100, 1000, 10000, 100000\}$. We report the simulation results from each scenario in the following.

Scenario I. Fig. 3.3(a) plots the average goodput gain and processing speed with different marker numbers. The processing speed is measured in Mbps and gathered from a commodity 2.6 GHz AMD workstation. This figure reveals that although more markers slightly increase the goodput gain, more markers also lead to higher computation overhead. From our simulation results, the processing speed for $m = 10$ is 230 Mbps and $m = 20$ is 80 Mbps. Hence, we recommend $m < 20$ to keep up with the Ethernet line speed.

Scenario II. We report the average goodput gain with different selection policy β in Fig. 3.3(b), which reveals that $\beta = 0.1$ results in the highest goodput gain. Hence we recommend $\beta = 0.1$.

Scenario III. Fig. 3.3(c) plots the average goodput gain and downlink overhead under different update frequency f. We estimate the downlink overhead by computing the *down/up ratio* between the downlink traffic amount over uplink traffic amount. This figure reveals that while small f results in higher goodput gain, it also leads to higher down/up ratio. Hence, we recommend $f = 1,000$. This gives a reasonable down/up ratio of 8.97, which is roughly in-line with many ISP's asymmetric access plans, e.g., Verizon's DSL plan [30] offers a downlink of 7.1 Mbps to 15 Mbps with an uplink of 768 Kbps.

In summary, we recommend $m = 10$, $\beta = 0.1$, and $f = 1,000$ for the best tradeoff between performance (uplink goodput gain) and overhead (processing speed and up/down ratio). If not otherwise specified, we use this configuration for following simulations. Last, we note that the uplink goodput gains shown in Figs. 3.3(a), 3.3(b), and 3.3(c) are

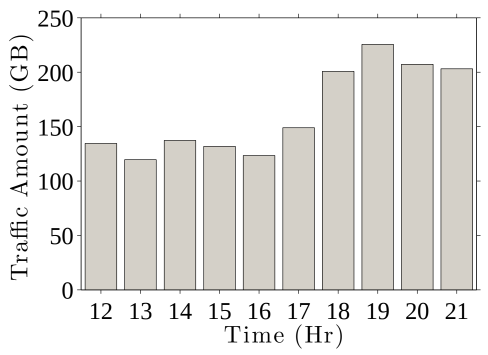


Figure 3.1: Traffic amount from 12:00 to 22:00.

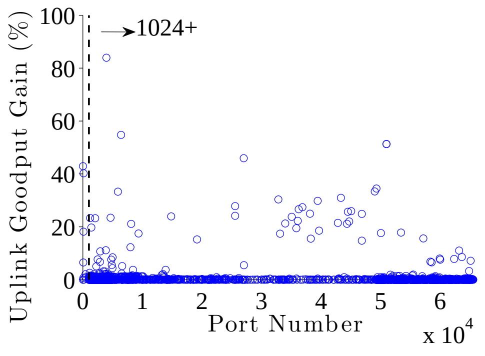
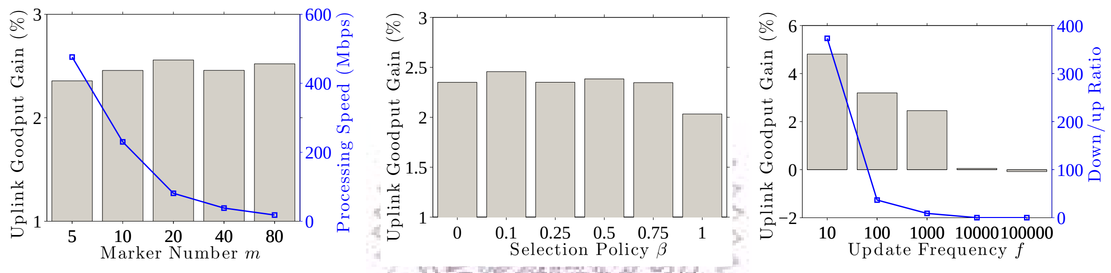


Figure 3.2: Mean uplink goodput gain on each port number, with 16MB cache size.



(a) (b) (c)

Figure 3.3: Implications of the system parameters: (a) marker number m, (b) selection policy β, and (c) update frequency f.

not as high as we observed in earlier sections. This is understandable because these figures present the average goodput gain over *all* data streams. The observation emphasizes the importance of identifying those data streams with more data redundancy, so that we can allocate more resources to them for higher overall goodput gain. In the next section, we present a detailed prediction model for this purpose.

3.2 Dynamic Adaptation Algorithm

In this section, we use the configuration recommended above to build the prediction model to estimate the goodput gain. With the prediction model, we design an adaptation algorithm which dynamically invests cache size to each data stream.

3.2.1 Prediction Model

We develop a prediction model that allows us to estimate the goodput gain by analyzing a few packets at the beginning of each data stream. We employ the aforementioned 8,147 sample data streams to drive our simulator. We study the relation between goodput gain and individual data stream features, under various cache size $B = B_r = B_s \in \{1, 2, 4, 8, 16, 32, 64\}$.

We first analyze the implication of source port number on uplink goodput gain, and plot the average goodput gain of each port number in Fig. 3.2. Note that, ports between 0 and 1,023 are called *well-known* ports, which are associated with the default protocols. Fig. 3.2, however, reveals that some higher ports ≥ 1024 also achieve high goodput gain. The actual protocols used on these high ports are unknown to us, and thus we cannot predict the goodput gain of many high port numbers. While Deep Packet Inspection (DPI) may help us to infer the protocols used by the data streams, DPI incurs non-trivial overhead [2, 26] and thus we do not consider it in this article.

Features other than port number are continuous, and thus we group the feature values into 10 bins and calculate the mean of each bin for better mathematical tractability. We perform linear and quadratic regressions on the mapping between the feature values and mean goodput gains. Table 3.1 shows the R^2 values of single-variable regression with cache size of 16 MB. This table shows that entropy, the ASCII ratio, and the mean packet length significantly outperform the standard deviation of packet length; therefore, we no longer consider the latter feature. We then perform two-variable regressions with a cache size of 16 MB, and report the R^2 values in Table 3.2. This table indicates that all the R^2 values of two variable regression are much lower than that of the single-variable quadratic regression of entropy. Thus, we use quadratic regression of entropy to build the prediction model.

Fig. 3.4 shows the distribution of uplink goodput gain with different entropy. In particular, we propose an empirical model to predict the uplink goodput gain of each data stream based on entropy H and cache size B. We mathematically write the model as:

$$\gamma(B, H) = \delta_{B,2}H^2 + \delta_{B,1}H + \delta_{B,0}, \tag{3.1}$$

where $H \in \{0, 2.66, 5.31, 7.07, 10.62, 13.28, 15.93, 18.59, 21.24, 23.90\}$, $B = B_r = B_s \in \{2, 4, 8, 16, 32, 64\}$, and $\delta_{B,2}$, $\delta_{B,1}H$, and $\delta_{B,0}$ are model parameters derived from the quadratic regression. For cache sizes and entropy from the sampled values, we use linear interpolation or extrapolation to approximate the goodput gain. Table 3.3 gives the model parameters. The average R^2 across all cache size is 0.864, which is fairly accurate. For visual validation, we plot sample actual and predicted goodput gains in Fig. 3.6, and the interpolated surface of the predication model in Fig. 3.7. Fig. 3.6 shows that our prediction

Table 3.1: R^2 value of single-variable regression with a 16MB cache

Feature	**Linear**	**Quadratic**
Entropy H	0.74	0.85
ASCII ratio θ	0.08	0.67
Standard diviation of packet length	0.04	0.16
Mean packet length	0.28	0.29

Table 3.2: R^2 value of two-variable regression with a 16MB cache

Feature	**Linear**	**Quadratic**
Entropy H and ASCII ratio θ	0.39	0.59
Entropy H and Mean packet length	0.45	0.61
ASCII ratio θ and Mean packet length	0.39	0.48

model closely follows the actual results, while both figures confirm that: (i) lower entropy leads to higher goodput gain and (ii) larger cache size results in higher goodput gain.

3.2.2 Adaptation Formulation and Algorithm

The simulation results presented above reveal that for a given data stream, larger cache size results in higher goodput gain. However, real systems always have limited memory size, and how to optimally allocate the memory to multiple concurrent data streams become a challenging issue. We let S be the total number of data streams and B_T be the total cache size. We use H_s to denote the sampled entropy of data stream s, where $s = 1, 2, \cdots, S$. Our problem is to find the best way to distribute B_T among all S data streams in order to maximize the overall goodput gain. We let b_s, where $s = 1, 2, \cdots, S$ be the allocated cache size of data stream s, which are the decision variables of our problem. We use $\sum_{s=1}^{S} \gamma(b_s, H_s)$, the overall goodput gain, as our maximization goal.

To formulate the optimization problem, we need to mathematically write the estimated goodput gain of data stream s as a function $\hat{\gamma}(\cdot)$ of the decision variable b_s under a given sampled entropy H_s. The prediction model in Eq. (3.1), can be written as a piecewise linear function. We let Z be the total number of endpoints of this piecewise linear function, and write the endpoints as $(d_1, g_1), (d_2, g_2), \cdots, (d_z, g_z)$, where the cache size $d_1, d_2, \cdots, d_z$ come from the chosen B in the simulations and the predicted goodput gain $g_{z,H_s} = \gamma(d_z, H_s), \forall z = 1, 2, \ldots, Z$.

We plot two sample piecewise linear functions with $H_s = 5.31$ and 10.62 in Fig. 3.5 for illustrations. We make an important observation: the slopes of individual segments are strictly decreasing. That is, $\frac{g_{z,H_s} - g_{z-1,H_s}}{d_{z,H_s} - d_{z-1,H_s}} > \frac{g_{z+1,H_s} - g_{z,H_s}}{d_{z+1,H_s} - d_{z,H_s}}$ for all $z = 2, 3, \cdots, Z-1$, and any H_s. Table 3.4 shows the parameters of the piecewise linear functions. We discard

Table 3.3: Prediction Model Parameters

δ	$B = 2$	$B = 4$	$B = 8$	$B = 16$	$B = 32$	$B = 64$
$\delta_{B,2}$	0.089	0.086	0.080	0.077	0.076	0.075
$\delta_{B,1}$	-3.351	-3.342	-3.256	-3.210	-3.165	-3.130
$\delta_{B,0}$	30.465	31.618	32.427	32.753	32.763	32.680

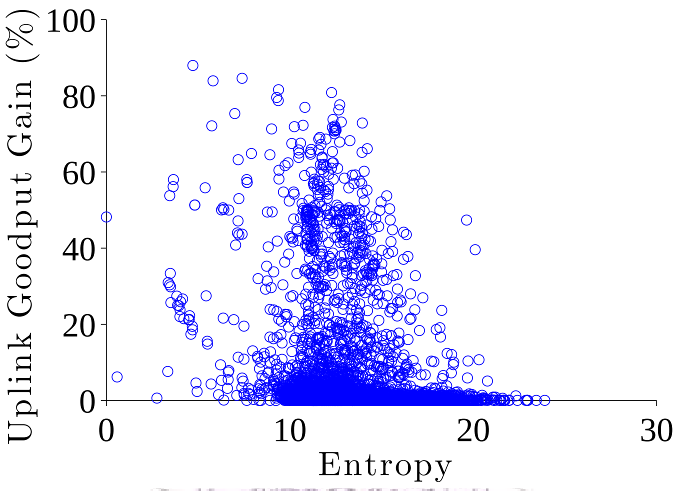


Figure 3.4: Distribution of uplink goodput gain for all data streams, with 16MB cache size.

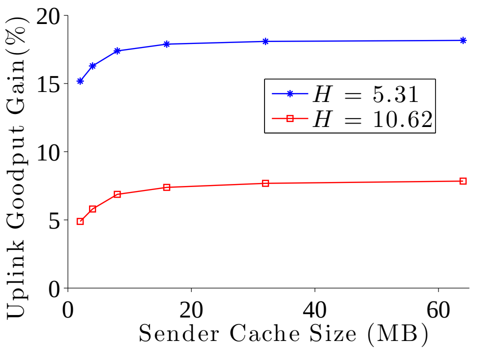


Figure 3.5: The piecewise linear functions with H_s = 5.31 and 10.62.

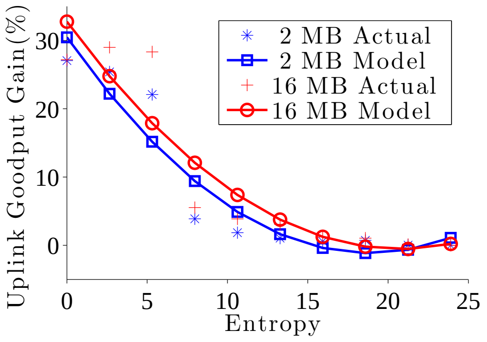


Figure 3.6: The prediction model closely follows the actual goodput gain.

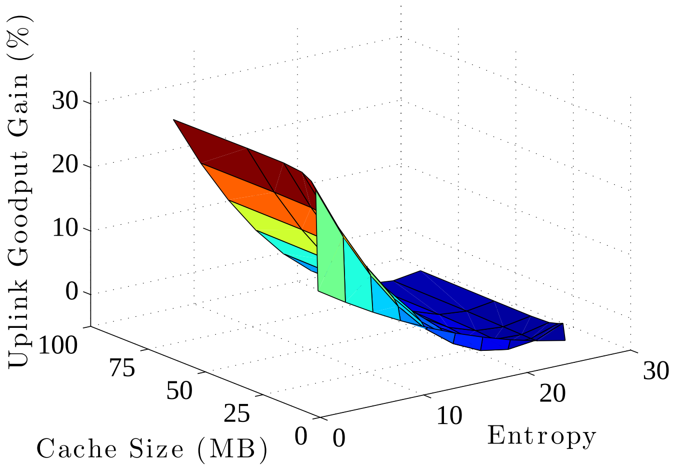


Figure 3.7: The interpolated surface of our proposed prediction model.

entropy $>$ 15.933 because of its low goodput gain (less than 3 %). Thus, we have 6 entropy bins. To validate this property that the slopes of individual segments are strictly decreasing, we check all 4×6 inequalities (since we have $Z = 6$ and 6 entropy bins), and found only 1 out of the 24 inequalities do not hold (Table 3.5) because the large cache size is enough to hold the redundancy data. Next, we divide each decision variable b_s into $Z - 1$ intermediate variables, composed of the $Z - 1$ segments. In particular, we define $0 \leq \hat{b}_{s,z} \leq d_{z+1} - d_z$, where $z = 1, 2, \cdots, Z - 1$ as the intermediate variables, and

$$b_s = \sum_{z=1}^{Z-1} \hat{b}_{s,z}. \tag{3.2}$$

Last, since the uplink goodput gain increases dramatically when the per data stream cache size is small, we reserve an *initial cache size* r_s for data stream s. We let the total initial cache size $R = \sum_{s=1}^{S} r_s$. Via extensive simulations, we empirically found that $r_s = 128$ KB leads to high uplink goodput gain. Hence, we set $r_s = 128$ KB if not otherwise specified.

With the notations defined above, we write the optimization problem as:

$$\hat{b}^{\star}_{s,z} = \operatorname{argmax} \sum_{s=1}^{S} \sum_{z=1}^{Z-1} \hat{b}_{s,z} l_{z,H_s}; \tag{3.3}$$

$$s.t. \sum_{s=1}^{S} \sum_{z=1}^{Z-1} \hat{b}_{s,z} \leq B_T - R; \tag{3.4}$$

$$0 \leq \hat{b}_{s,z} \leq d_{z+1} - d_z, \tag{3.5}$$

where $l_{z,H_s} = \frac{g_{z+1,H_s} - g_{z,H_s}}{d_{z+1,H_s} - d_{z,H_s}}$ is the segment slope computed from Eq. (3.1). The formulation is Eqs. (3.3)–(3.5) is a Linear Programming (LP) problem and can be solved by

Table 3.4: Piecewise Linear Function Parameters

Entropy H	$B_s = 2MB$	$4MB$	$8MB$	$16MB$	$32MB$	$64MB$
0	30.465	31.618	32.427	32.753	32.763	32.680
2.656	22.192	23.348	24.345	24.774	24.893	24.893
5.311	15.172	16.287	17.392	17.886	18.087	18.158
7.067	9.403	10.438	11.568	12.088	12.348	12.475
10.622	4.888	5.798	6.873	7.382	7.674	7.843
13.278	1.624	2.368	3.307	3.766	4.067	4.264
15.933	-0.387	0.149	0.870	1.241	1.524	1.736
18.589	-1.145	-0.860	-0.438	-0.194	0.048	0.260
21.244	-0.651	-0.659	-0.617	-0.538	-0.363	-0.164
23.900	1.096	0.753	0.334	0.210	0.292	0.464

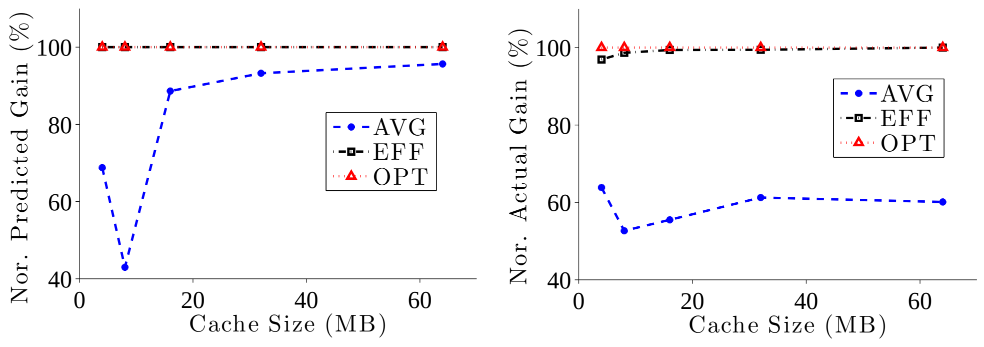
Table 3.5: Slopes of the Piecewise Linear Functions

Entropy H	$l_{2MB,H}$	$l_{4MB,H}$	$l_{8MB,H}$	$l_{16MB,H}$	$l_{32MB,H}$
0	0.576	0.202	0.041	0.001	-0.003
2.656	0.578	0.249	0.054	0.007	0.000
5.311	0.558	0.276	0.062	0.013	0.002
7.067	0.517	0.283	0.065	0.016	0.004
10.622	0.455	0.269	0.064	0.018	0.005
13.278	0.372	0.235	0.057	0.019	0.006

various LP solvers, such as CPLEX [11]. Once the optimal solution $\hat{b}^{\star}_{s,z}$ is computed, the optimal allocation $b^{\star}_{s}$ can be derived using Eq. (3.2). We refer to this optimal algorithm as OPT throughout this thesis. We emphasize that we can successfully allocate resources because the segment slopes are decreasing, and thus an optimal LP solution always satisfies $\hat{b}^{\star}_{s,z+1} > 0 \Rightarrow \hat{b}^{\star}_{s,z} = d_{z+1} - d_z$, for any $z = 1, 2, \cdots, Z-1$. In other words, an LP solver would not invest any memory on segment $z+1$ unless no more memory can be allocated on segment z.

3.3 Large-Scale Simulations and Evaluations

We use large-scale simulations to evaluate the performance of the adaptation algorithm presented above.



(a) (b)

Figure 3.8: Goodput gain, normalized to OPT: (a) predicted and (b) actual.

3.3.1 Evaluations on Adaptation Algorithm

Since solving LP problems may be time consuming, we propose an efficient algorithm, EFF, which runs in real time. At each iteration, the EFF algorithm invests the remaining cache size on the data stream that is estimated to achieve the highest goodput gain. The algorithm runs until the remaining cache size reaches zero. To evaluate its performance, which we have also implemented two other adaptation algorithms for comparisons. AVG: equally divides the total cache size to each data stream, and OPT, which is the CPLEX-based optimal algorithm.

The adaptation algorithms are triggered periodically every U packets. We set the adaptation period $U = 1,000,000$ in the simulations. We only consider the hosts with 8+ data streams, because the adaptation is less meaningful on hosts with very few data streams.

Goodput gain. We first report the expected uplink goodput gain achieved by different algorithms. The average goodput gain achieved by OPT is between 1.57% and 5.98%, and Fig. 3.8(a) gives the goodput gain normalized to that of OPT. This figure reveals that EFF achieves very similar goodput than that of OPT, and outperforms AVG by up to 45%. Fig. 3.8(b) reports that actual goodput gain computed by the simulator. We make two observations on Fig. 3.8(a) and 3.8(b): (i) the trends of goodput gain are consistent and (ii) our prediction model is conservative and over-estimates the goodput gain of the AVG algorithm (Fig. 3.8(a)); in reality the AVG algorithm results in much lower goodput gain (Fig. 3.8(b)).

Overhead. We next present the overhead of different algorithms. Fig. 3.9 shows the computational overhead, which is the average running time normalized to that of AVG. This figure reveals that our EFF algorithm runs as fast as AVG, while OPT may take a much longer time to terminate. The OPT algorithm also consumes much more memory as indicated in Fig. 3.10. More specifically, OPT may consume more than 20 MB memory,

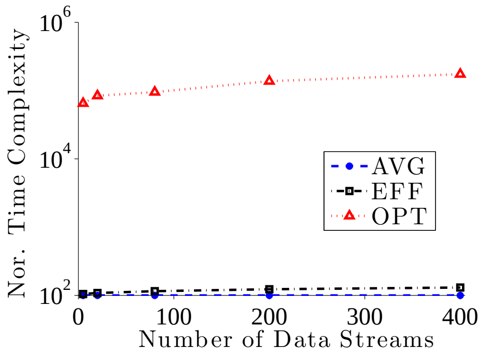


Figure 3.9: Time complexity, normalized to AVG.

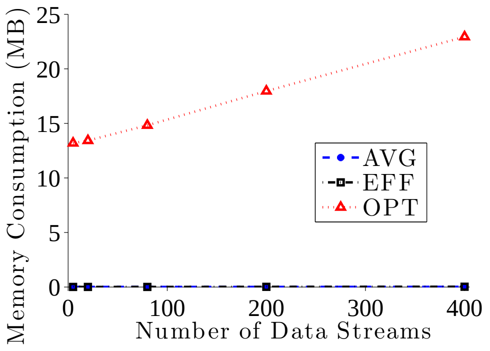


Figure 3.10: Memory consumption under different number of data streams.

which is nontrivial to some platforms, such as application routers. Given the high time and space overhead of the OPT algorithm, we no longer consider it in the rest of this article.

Comparisons against symmetric redundancy elimination algorithm The strengths of RCARE compared to the current redundancy elimination algorithm are (i) it can adjust the arrangement of cache sizes for data streams on the fly, (ii) it can leverage storage and computation overhead from a sender to a receiver, and (iii) the cache on a receiver can support inter-sender redundancy elimination which is our future work. In order to achieve these three goals, we sacrifice little goodput gain. We take sample data streams on number of data streams from 2 to 7 since 93% data streams falling in this range. We let $B_s = 4$, $B_r = 16$, and vary $f \in \{5, 50, 100, 500, 1000, 2000\}$. Fig. 3.12 presents the goodput gains on different update frequency f. Although we sacrifice little goodput gain on switching from EndRE to RCARE, the results reveal that the accuracy of selection policy is high. We only have little degrade on goodput gain with the growing f. The maximum goodput gain in EndRE is 95.06% and 80.59% in RCARE.

3.3.2 Performance Gain of RCARE

We quantify the benefits of RCARE using extensive traces under two different deployment configurations: (a) *host based*, in which each host maintains a cache for each data stream, and (b) *proxy based*, in which proxy maintains a cache for each data stream. For proxy based deployment, we regard each individual source port as different data streams. Fig. 3.13 illustrates these configurations. We use the configuration recommended in Sec. 3.1.3, and let $B_r = 16$, $\alpha = 1,000,000$, and $U = 1,000,000$. We have imple-

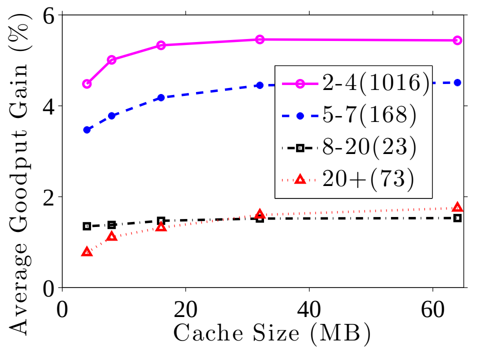


Figure 3.11: Number of data streams and goodput gain.

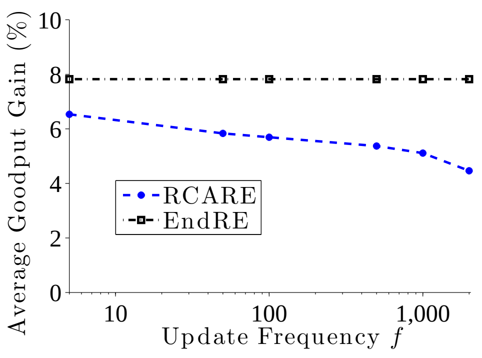


Figure 3.12: Goodput gain on RCARE and EndRE.

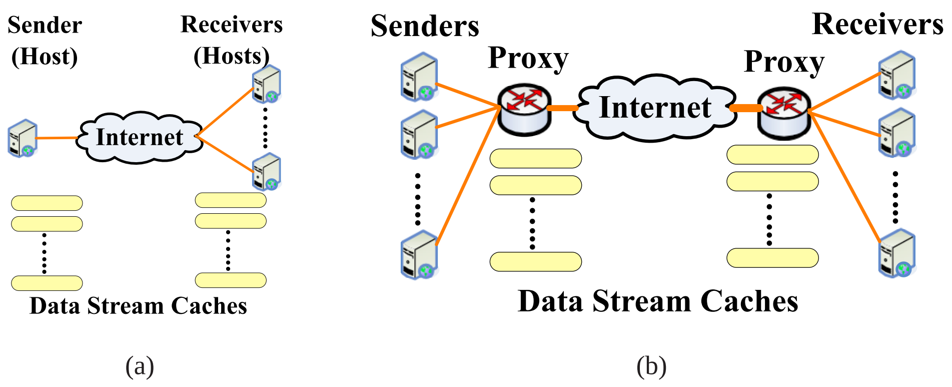


Figure 3.13: Deployment configurations: (a) host based and (b) proxy based.

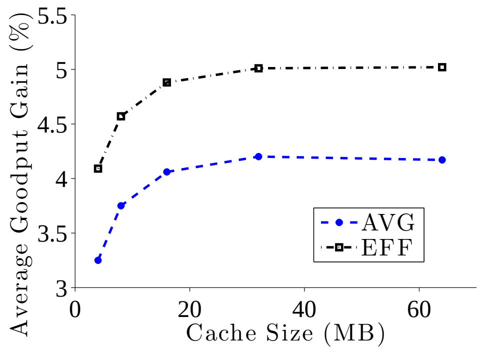


Figure 3.14: Host with average goodput gains.

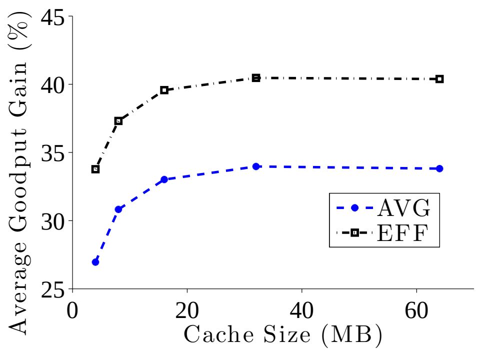


Figure 3.15: The first 10% hosts with average goodput gain.

Table 3.6: Proxy Based Goodput Gain

Algorithm	$B = 0.25\,GB$	$0.5\,GB$	$1\,GB$	$4\,GB$	$16\,GB$
AVG	0.13%	0.12%	0.12%	0.12%	0.13%
EFF	0.60%	1.12%	1.78%	2.66%	2.87%

mented the RCARE with the EFF algorithm, and compare it against the AVG algorithm.

Host based. We vary $B_s \in \{4, 8, 16, 32, 64\}$. We use the 10-hour traces. Since we want to compare the performance between the two adaptation algorithms, we only consider the the number of data streams ≥ 2 on each host. There are 876 GB trace data and 1,280 hosts in total. We calculate the average goodput gains of data streams on each host. Fig. 3.14 shows the average goodput gain of all hosts with different cache sizes. It shows that EFF always outperforms AVG. For example, the EFF algorithm with 4 MB cache size achieves almost the same goodput gain of the AVG algorithm with 32 MB cache size. Fig. 3.11 presents the relation between the number of data streams and goodput gains. Each piecewise line represents by the [number of data streams in the host(amount)]. There are 93% data streams falling in the range [2,7]. They have at least 3.47% goodput gain in average. We zoom in to the first 10% hosts achieving the highest goodput gain, and plot it in Fig. 3.15. This figure shows that the EFF algorithm achieves over 40% goodput gain on average.

Proxy based. We let $f = 10,000$ and vary $B_s \in \{0.25, 0.5, 1, 4, 16\}$ GB. We consider 21.6 GB sample traces, which contain 8,354 data streams. The average goodput gain is given in Table 3.6. This table shows that the uplink goodput gains of EFF are at least 10 times and at most 22 times higher than that of AVG.

In summary, via extensive trace-driven simulations, we demonstrate that RCARE achieves much higher uplink goodput gain with the EFF adaptation algorithm, which runs in real-time. Hence, we recommend using the EFF algorithm.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

In this thesis, we proposed a practical asymmetric communication algorithm, called RCARE, for bandwidth and capability asymmetric communications. RCARE is among the first practical network algorithms to maximize the uplink goodput gain of resource-constrained senders by leveraging the already deployed downlink bandwidth and receiver capability. Our extensive simulation results reveal the merits of RCARE, compared to existing asymmetric communication algorithms [3, 10, 13, 14, 22, 32], RCARE achieves much higher uplink goodput gain: up to 50 times improvement is possible, and much lower relative overhead on downlink traffic: up to 384 times reduction is observed. This shows that RCARE successfully improves the uplink goodput gain over the existing asymmetric communication algorithms, such as ListQuery [13, 14], while incurring small downlink traffic overhead.

RCARE is also different from the state-of-the-art redundancy elimination algorithms [4, 6, 16, 18, 24, 29, 33, 35] in several aspects. First, RCARE leverages the idling downlink bandwidth and receiver capability for higher uplink goodput gain. Second, RCARE shifts computational and storage complexity from the sender to receiver, e.g., per-packet encoding time at the sender is only 0.5 msec, while the decoding time at the receiver is about 5 msec. This shows that RCARE is suitable to the considered usage scenarios illustrated in Fig. 1.1, while protocol-independent redundancy elimination algorithms [4, 5, 27, 35] dictate the same cache size on the server and receiver, and thus cannot utilize additional resources at powerful receivers to help resource-constrained senders. Third, RCARE is flexible on cache size adaptation. In real life, we sometimes run a heavy applications on resource-constrained senders. We may temporarily shift some reserved resources from RCARE to other applications; and shift the resources back once the other applications are finished.

Moreover, RCARE is adaptive. We analyze features with sample data streams and use it to derive prediction models. The analysis helps us to optimally allocate cache size among the data streams for higher uplink goodput gain. Our proposed adaptation algorithm further enhances the performance of RCARE: (i) it improves 87% uplink goodput gain compared to a baseline of equal division and (ii) top 10% data streams achieve up to 40% uplink goodput gain on average.

4.2 Future Work

There are several future research directions, for example:

- RCARE has potential to support inter-sender redundancy elimination, which is crucial as most Internet services concurrently or sequentially support many clients. For such services, maintaining a shared cache at the receiver allows us to further increase the uplink goodput gain and save the computational overhead. Determining the relevance among multiple data streams is one of our future tasks.
- User have different behavioral patterns. We may create multiple versions of cache, called *behavior caches*, and use them in different contexts, e.g., two separate caches may be used for weekdays and weekends, respectively. Incorporating behavior caches to RCARE is another future task.
- We believe the RCARE can run in real time after some code optimizations. We plan to implement RCARE in a real network stack and conduct actual experiments to demonstrate this.
- Cellular ISPs are switching away from unlimited data plans, and thus the total network traffic amount (in both directions) becomes a key concern. Adjusting the update frequency f in order to minimize the total traffic amount is one of our future tasks.
- Some of the RCARE senders may be battery-powered, which have limited energy budget. We will measure the energy overhead of RCARE on these senders.

Bibliography

[1] WAN and application optimization solution guide. `http://www.cisco.com/en/US/docs/nsite/wan_optimization/WANoptSolutionGd.pdf`, April 2008.

[2] G. Aceto, A. Dainotti, W. De Donato, and A. Pescape. Portload: Taking the best of two worlds in traffic classification. In *Proc. of IEEE INFOCOM'10*, pages 1 –5, march 2010.

[3] M. Adler and B. Maggs. Protocols for asymmetric communication channels. *Journal of Computer and System Sciences*, 63(4):573–596, December 2001.

[4] 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 USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*, pages 419–432, San Jose, CA, June 2010.

[5] 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 ACM SIGCOMM'08*, pages 219–230, Seattle, WA, August 2008.

[6] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee. Redundancy in network traffic: Findings and implications. In *Proc. of Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance'09)*, pages 37–48, Seattle, WA, June 2009.

[7] B. Aslam, P. Wang, and C. Zou. Pervasive Internet access by vehicles through satellite receive-only terminals. In *Proc. of IEEE International Conference on Computer Communications and Networks (ICCCN'09)*, San Francisco, CA, August 2009.

[8] L. Atzori, A. Iera, and G. Morabito. The Internet of things: A survey. *Computer Networks*, 54(15):2787–2805, October 2010.

[9] H. Balakrishnan and V. Padmanabhan. How network asymmetry affects TCP. *IEEE Communications Magazine*, 39(4):60–67, April 2001.

[10] P. Bose, D. Krizanc, S. Langerman, and P. Morin. Asymmetric communication protocols via hotlink assignments. *Theory of Computing Systems*, 36(6):655–661, November 2003.

[11] IBM ILOG CPLEX Optimizer. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.

[12] M. Dischinger, A. Haeberlen, K. Gummadi, and S. Saroiu. Characterizing residential broadband networks. In *Proc. of ACM SIGCOMM Internet Measurement Workshop (IMC'07)*, pages 43–56, San Diego, CA, October 2007.

[13] T. Gagie. Dynamic asymmetric communication. In *Proc. of International Colloquium on Structural Information and Communication Complexity (SIROCCO'06)*, pages 310–318, Chester, UK, July 2006.

[14] T. Gagie. Dynamic asymmetric communication. *Information Processing Letters*, 108(6):352–355, November 2008.

[15] E. Halepovic, C. Williamson, and M. Ghaderi. DYNABYTE: A dynamic sampling algorithm for redundant content detection. In *Proc. of IEEE International Conference on Computer Communications and Networks (ICCCN'11)*, pages 1–8, Maui, HI, July 2011.

[16] M. Hefeeda, C. Hsu, and K. Mokhtarian. Design and evaluation of a proxy cache for peer-to-peer traffic. *IEEE Transactions on Computers*, 60(7):964–977, July 2011.

[17] B. Jenkins. Algorithm alley: Hash functions. *Dr. Dobb's Journal*, September 1997.

[18] C. Lee and R. Yang. High-throughput data compressor designs using content addressable memory. *IEE Proceedings on Circuits, Devices and Systems*, 142(1):69–73, February 1995.

[19] Y. Li, C. Trang, X. Huang, C. Hsu, and P. Lin. CacheQuery: A practical asymmetric communication algorithm. In *Proc. of IEEE Global Telecommunications Conference (GLOBECOM'12)*, Anaheim, CA, December 2012.

[20] Y.-D. Lin, I.-W. Chen, P.-C. Lin, C.-S. Chen, and C.-H. Hsu. On campus beta site: architecture designs, operational experience, and top product defects. *IEEE Communications Magazine*, 48(12):83 –91, December 2010.

[21] M. Mathur. Elucidation of upcoming traffic problems in cloud computing. *Recent Trends in Networks and Communications: Communications in Computer and Information Science*, 90(1):68–71, July 2010.

[22] G. Mazzini. Asymmetric channel cooperative compression. *IEEE Communications Letters*, 12(4):328–330, April 2008.

[23] I. Minei and R. Cohen. High-speed Internet access through unidirectional geostationary satellite channels. *IEEE Journal on Selected Areas in Communications*, 17(2):345–359, January 2004.

[24] D. Munteanu and C. Williamson. An FPGA-based network processor for IP packet compression. In *Proc. of SCS International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'05)*, pages 599–608, Philadelphia, PA, July 2005.

[25] The network simulator. `http://www.isi.edu/nsnam/ns/`.

[26] B.-C. Park, Y. Won, M.-S. Kim, and J. Hong. Towards automated application signature generation for traffic identification. In *Proc. of Network Operations and Management Symposium (NOMS'08)*, pages 160 –167, april 2008.

[27] N. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proc. of ACM SIGCOMM'00*, pages 87–95, Stockholm, Sweden, August 2000.

[28] C. M. Trang, X. Huang, and C.-H. Hsu. Pushing uplink goodput of an asymmetric access network beyond its uplink bandwidth. In *Proc. of IEEE International Conference on Communications (ICC'12)*, Ottawa, Canada, June 2012.

[29] C. Tye and D. Fairhurst. A review of IP packet compression techniques. In *Proc. of Annual PostGraduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet'03)*, Liverpool, UK, June 2003.

[30] Verizon DSL Internet Plans. `http://www22.verizon.com/home/highspeedinternet/#plans`.

[31] R. Wang, T. Taleb, A. Jamalipour, and B. Sun. Protocols for reliable data transport in space Internet. *IEEE Communications Surveys and Tutorials*, 11(2):21–32, Second Quarter 2009.

[32] J. Watkinson, M. Adler, and F. Fich. New protocols for asymmetric communication channels. In *Proc. of International Colloquium on Structural Information and Communication Complexity (SIROCCO'01)*, pages 337–350, Catalonia, Spain, June 2001.

[33] R. Yang and C. Lee. High-throughput data compressor designs using content addressable memory. In *Proc. of IEEE International Symposium on Circuits and Systems (ISCAS'94)*, pages 147–150, London, UK, May 1994.

[34] zlib home page, April 2010. `http://zlib.net/`.

[35] E. Zohar, I. Cidon, and O. Mokryn. The power of prediction: Cloud bandwidth and cost reduction. In *Proc. of ACM SIGCOMM'11*, pages 86–97, Toronto, Canada, August 2011.

[36] M. Zorzi, A. Gluhak, S. Lange, and A. Bassi. From today's INTRAnet of things to a future INTERnet of things: A wireless- and mobility-related view. *IEEE Wireless Communication Magazine*, 17(6):44–51, December 2010.

Symbol Table

Symbol	Description
B_r	Receiver cache size
B_s	Sender cache size
f	Update frequency
β	Selection policy parameter
P_β	Selection policy
w	Representative window size
p	Representative window sampling frequency
m	Marker list size
T_m	Marker list refresh threshold
r_m	Marker list update frequency
θ	ASCII ratio
H	Entropy
α	Number of sampling blocks for data stream features
S	Total number of data streams
s	Data stream
B_T	Total cache size
b	Allocated cache size
Z	Total number of endpoints of piecewise linear function
z	Endpoint of piecewise linear function
d	Cache size on endpoint of piecewise linear function
g	Goodput gain on endpoint of piecewise linear function
R	Total initial cache size
r	Initial cache size
l	Segment slope
U	Adaptation period