Hash challenges: Stretching the limits of compare-by-hash in distributed data deduplication ☆

João Barreto *, Luís Veiga, Paulo Ferreira

INESC-ID and Technical University of Lisbon, Portugal

ARTICLE INFO

ABSTRACT

We propose a technique for reducing communication overheads when sending data across a network. Our technique, called hash challenges, leverages existing deduplication solutions based on compare-by-hash by being able to determine redundant data chunks by exchanging substantially less meta-data. Hash challenges can be used directly on any existing compare-by-hash protocol, with no relevant additional computational complexity. Using real data from reference workloads, we show that hash challenges can save as much as 64% meta-data exchanged across the network, relatively to plain compare-by-hash. This implies reductions of up to 7% in overall transferred volume, and performance gains of up to 16% with typical asymmetrical broadband connections.

1. Introduction

Many interesting and useful applications require transferring large volumes of data across a network, from network file and backup systems, content delivery networks and software distribution mirroring systems, to recently popular cloud-based file hosting and sharing systems [1]. While broadband, both wired and mobile, is rapidly becoming ubiquitous in developed countries [2], it is not a panacea. Firstly, upload bandwidth remains a scarce resource for the average broadband consumer, for commercial and technical reasons that are unlikely to change in the near future [2]. Yet, in many examples mentioned above, clients tend to upload more than they download; e.g., in backup and file hosting/sharing systems. Furthermore, many ISPs charge a per-byte price, whereas network usage incurs energy costs to battery-constrained mobile clients. Finally, broadband penetration is still far from significant in the developing world [3].

Fortunately, as many studies show (e.g., [4,5]), the data exchanged between hosts in most of these systems exhibits high levels of redundancy. Hence, much of the traffic that the above mentioned applications exchange can be eliminated. To illustrate, consider the situation where some node (the *sender*) is about to send file S to another node (the *receiver*). Assume that the receiver already stores a set of files, R. Let us call each contiguous data block in R and S a *chunk*. Very frequently, S will share many chunks with other files (or versions of files) that the receiver already holds (in R). If the sender is able to infer which chunks in S are redundant across R, then it can avoid transferring them, as the receiver can directly obtain them within the local contents in R.

This problem is often called *distributed data deduplication* (hereafter simply called *deduplication*). Much recent work has proposed different deduplication techniques [5], the most prominent being the *compare-by-hash* method (CBH) [6,4]. In CBH, the sender starts by dividing each file to be sent into a sequence of contiguous chunks, $c_1, c_2, \ldots, c_n$, and by computing the corresponding hashes, using some hash function with a negligible collision probability (e.g. SHA-1 [7]). The sender then transmits the hash list to the receiver. The receiver maintains a lookup structure on the hash of each block in R, usually called a *chunk hash database*. Upon receiving the hash list from the sender,

the receiver searches for each hash in its local chunk hash database. Whenever a match is found, the receiver retrieves the contents from the local chunk (assuming the local chunk to be identical to the original chunk at S, given the negligible collision probability). The receiver then tells the sender which chunks in S are still missing, so that the sender can transfer such chunks only.

The power of CBH lies in its ability to detect any cross-file and cross-version redundancy, while requiring no long-term shared state between sender and receiver. However, CBH's effectiveness depends on using chunks that are small enough to exploit most fine-grained redundancy that S shares with R. Unfortunately, precision in CBH has a high price in terms of latency and network overhead of the hash exchange phase. As one attempts to leverage precision by reducing chunk sizes, the meta-data overhead will easily outgrow the savings in the second protocol round [8]. Not surprisingly, most deployed CBH systems are forced to run at a relatively coarse precision (e.g., 2 Kbytes expected chunk size in LBFS [4]), thus neglecting substantial redundancy.

This paper proposes a lighter alternative to CBH, called *Hash Challenges* (HCs). HCs stretch CBH's limits by dramatically reducing the underlying meta-data overhead, while detecting redundant chunks as effectively as the latter (i.e., for identical chunk sizes, HCs detect the same redundancy as CBH). Intuitively, such savings arise from the ability of HCs to filter out most non-redundant chunks by exchanging very short hash fragments, instead of their complete hashes. HCs can directly replace the CBH steps in any existing protocol based on CBH, including the more intricate CBH variations that have been proposed recently (e.g. [8–11]). Moreover, HCs introduce neither any relevant computation overhead nor network round-trips relatively to CBH.

We support the above claims with a theoretical analysis and an experimental evaluation of the HCs protocol. Running a full-fledged implementation of HCs with standard workloads from reference work in data deduplication, we show that HCs can save as much as 64% meta-data exchanged across the network, relatively to CBH. Such reductions mean gains in overall transferred volume of up to 7%, with no relevant performance degradation.

Most importantly, the benefits of HCs are amplified in scenarios where data to deduplicate is to be sent over a low bandwidth link. In a 1 Mbps uplink/100 Mbps downlink scenario, HCs are up to 16% faster than CBH.

The remainder of the paper is organized as follows. Section 2 describes and analyzes HCs. Section 3 provides experimental results that confirm the advantages of HCs over CBH. Finally, Section 4 discusses related work and Section 5 draws concluding remarks.

2. Hash challenges

HCs ensure the same as CBH: to allow the sender node to know whether the receiver already holds a given chunk, c, in its local repository, R. Fig. 1 depicts HCs. Like CBH, HCs work in three phases, which we explain next. Hereafter, we assume FIFO message ordering.
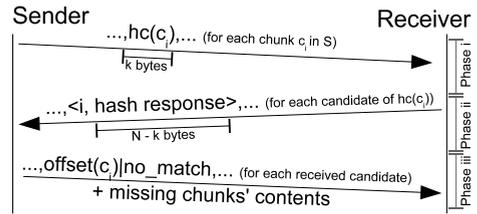
Fig. 1. Hash Challenges (HCs) protocol.

Phase i. The sender starts by dividing the contents to send (S) into a sequence of (contiguous) chunks. We can use any standard approach for defining chunk boundaries, either fixed- or variable-sized [5]. For each chunk c to transfer, the sender computes c's hash, $h(c)$, using some cryptographic hash function, e.g. SHA-1 [7]. Hereafter, let N denote the hash length in bytes (e.g. $N = 20$ for SHA-1).

Chunk division results in a list of meta-data entries that characterize each chunk in S, including the chunk's hash value, length and the address of its contents. The sender will maintain this list until phase *iii*.

While CBH would send the N-byte chunk hashes of every chunk in S to the receiver, our protocol will only send k-byte prefixes of each such hash. We call each prefix in the sequence sent to the receiver a *hash challenge*. The key insight behind HCs is that, even if the sender delivers such incomplete pieces of meta-data to the receiver, the sender will still be able to infer whether each chunk c in S is redundant across R or not. We explain how next.

Phase ii. The receiver maintains a local chunk hash database, which stores chunk meta-data for each chunk the receiver stores in R. For any given hash challenge, the local chunk hash database is able to lookup the chunks whose first k hash bytes match the given hash challenge. We designate such chunks as the *candidates* for the given hash challenge. Furthermore, we designate the remaining $N - k$ bytes of the chunk's hash as that candidate's *hash response* (to the HC).

For each received hash challenge $hc(c_i)$, the receiver inspects its local chunk hash database for one or more candidates that match $hc(c_i)$. For each candidate that the receiver finds for $hc(c_i)$, the receiver adds it to a *candidate list* and sends the candidate's hash response, labeled by the i index. The candidate list is held by the receiver until phase *iii*.

It is straightforward to extend the hash table-based chunk hash databases of most conventional CBH systems in order to support the above candidate lookup operation. A particularly efficient solution is to use a hash table with as many buckets as possible HCs (i.e. $2^{k \times 8}$). Each bucket i contains a list of meta-data entries for the chunks whose k-byte hash prefix equals i. Hence, finding the candidates for some hash challenge $hc(c_i)$ is as simple as directly returning all the entries in the bucket whose index is $hc(c_i)$.

Phase iii. Upon reception of each candidate's hash response for some hash challenge $hc(c_i)$, the sender obtains the real hash value of chunk c_i and compares both. If an effective match is found, then the receiver is actually holding the same chunk (assuming no hash collisions occur). In this case, the sender replies with the offset (within S) to
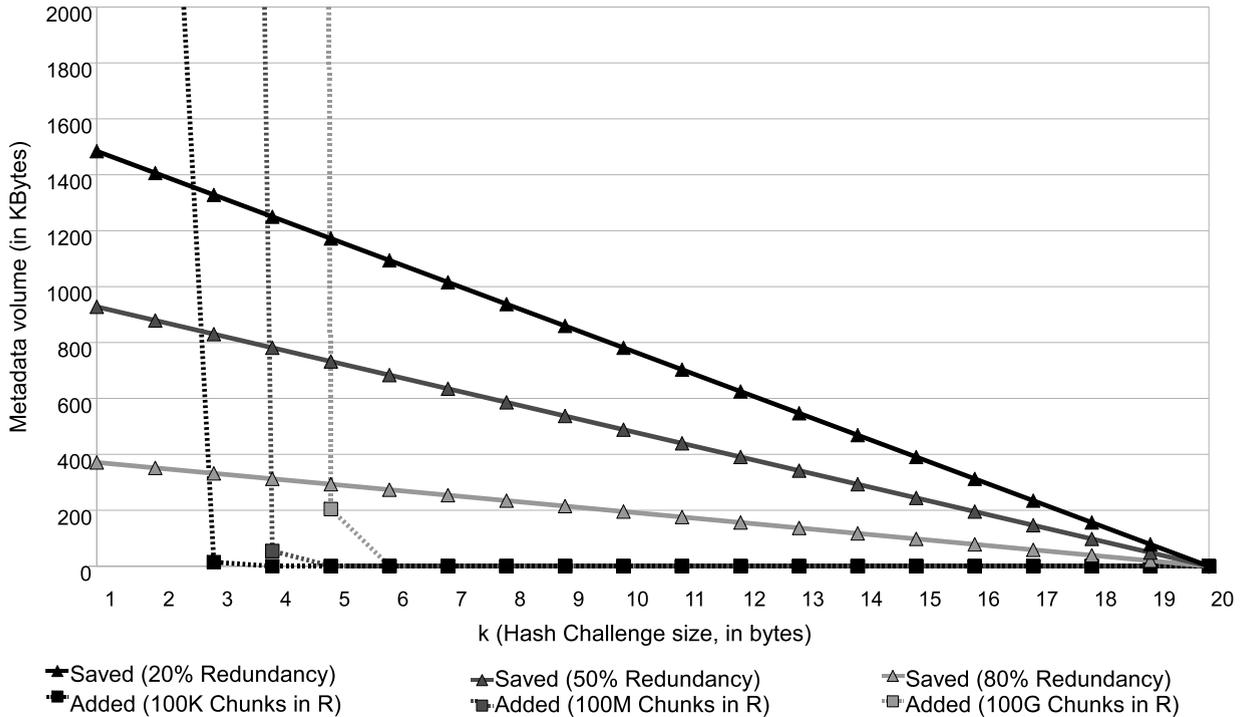
Fig. 2. Meta-data saved and added by HCs for different sizes of R and redundancy probabilities.

where the receiver must copy the redundant contents of the candidate chunk that the receiver holds.

If, otherwise, the hash response does not match the actual chunk hash, the current candidate is false. Hence, the sender replies with a reserved offset value (*no match*), telling the receiver to ignore the current candidate.

Finally, the sender transfers the chunks it did not find redundant. The receiver fills the non-redundant gaps with such contents, thereby completing S.

2.1. Analysis

It is easy to see that, in the absence of false candidates, HCs exchange less hash value meta-data than CBH for the non-redundant chunks. By only sending k bytes across the network and receiving an empty answer, the sender is immediately sure that the receiver does not hold c. Furthermore, in the case that the receiver does hold chunk c and no other chunk shares c's k-byte hash prefix, HCs exchange the same number of hash bytes as CBH: N bytes (k plus $N - k$, in HCs). Of course, since HCs save meta-data, their overall impact will only be noticeable when meta-data volume is non-negligible; as we show briefly, this is true for chunk sizes below 8 Kbytes.

It is worth noting that, by varying the size of hash challenges (from $k = N$ to 0) we can actually define a spectrum of protocols. At the $k = N$ extreme we get CBH. As we drop k, we achieve higher efficiency due to the above mentioned meta-data savings with non-redundant chunks. However, decreasing k will inevitably increase the likelihood of false candidates and consequently introduce network overhead.

We need to precisely predict the aggregate volume that HCs exchange in order to better understand this trade-off. We prove in Appendix A that the expected number of bytes transferred is bounded from above by:

$$\underbrace{\frac{|S|}{cs} \times k}_{\text{phase } i} + \underbrace{\#cand \times \left[(N - k) + int\right]}_{\text{phase } ii}$$

$$+ \underbrace{int \times \#cand + \left[1 - P_{red}(S, R, cs)\right] \times \frac{|S|}{cs} \times cs}_{\text{phase } iii}$$

where $|S|$ and $|R|$ denote the size (in bytes) of S and R, resp.; int denotes the size of an integer (used to express chunk indexes and offsets); $P_{red}(S, R, cs)$ denotes the probability that each chunk in S is redundant across R, assuming an average chunk size of cs; and $\#cand$ denotes the number of candidates found at the receiver on phase ii. The latter is given by: $\#cand = [P_{red}(S, R, cs) + \frac{|R|}{cs} \times \frac{1}{2^{k \times 8}}] \times \frac{|S|}{cs}$.

The above expression leads us to some fundamental conclusions about HCs. Firstly, starting at $k = N$ and gradually decreasing k, meta-data savings due to HCs grow much faster than the false candidate rate increases, until a critical threshold is reached. This is evident in Fig. 2, which depicts the meta-data volume that HCs actually save (by exchanging less hash bytes) and introduce (when false candidates occur).[1] For different combinations of two key

---

[1] The plotted values are estimates obtained from the components of the aggregate volume expression of HCs (see Appendix A for details).

variables, the size of R and the redundancy probability, the effective outcome of HCs is given by the vertical distance between both lines in Fig. 2. The higher the gain line is relatively to the loss line, the less the system will actually transfer across the network. For any scenario, such a positive difference tends to grow as one drops k. However, when one reaches the critical threshold (for instance, $k = 3$ for R with 100K chunks), such gains are almost instantaneously lost with an explosive increase of false candidates.

Hence, the success of deploying HCs depends on a careful choice of parameter k; more precisely, on choosing the *lowest k with a negligible false candidate rate*. As Fig. 2 shows, that choice depends only on the size of R. Any system based on HCs should thus either adapt k as R changes, or simply limit R to a maximum acceptable size and set k accordingly.

As Fig. 2 illustrates, smaller sizes of R maximize the effective gains that HCs can attain. This suggests that HCs are especially suited to workloads with smaller sizes of R. Nevertheless, the gains of HCs drop slowly as one considers much larger sizes of R (e.g. up to 100 TB with 1 KB chunks, k can be as low as 6 without incurring frequent false candidates).

In scenarios where the receiver stores a large repository, one option for using a low k is to partition that repository into smaller sub-domains, each one corresponding to a different R. This technique is not new. For instance, it is extensively used in Web content deduplication, where it is well studied that, for many Web workloads, by restricting redundancy detection to smaller content partitions (e.g. a single directory instead of the full repository at the server), similar redundancy levels are achieved [12].

Another crucial conclusion is that HCs *always* reduce meta-data volume on the sender-to-receiver link, while any additional traffic due to false candidates exclusively affects the receiver-to-sender link. This is particularly advantageous in asymmetric network where the bandwidth from the sender to the receiver is lower than in the opposite direction. For instance, most Internet broadband users face this situation when uploading new contents to a file sharing service. Besides reducing the overall volume, HCs also shift part of such volume to the higher bandwidth stream.

We empirically confirm these predictions next.

3. Evaluation

This section evaluates HCs, with the goal of answering two main questions: in practice, (1) *how much network volume do HCs save?*; and (2) *how do HCs impact performance?* To answer both questions, we built two file transfer systems: one that implements the CBH protocol used by LBFS [4], and an extension of the previous system that runs the HCs protocol. Both systems use Rabin fingerprints to delimit chunks using content-based boundaries, 20-byte SHA-1 hash values to identify chunks ($N = 20$), and maintain a 128M-bucket in-memory chunk hash table at the receiver site. The expected chunk size, hereafter denoted ecs, is a tunable parameter, and can be set from 128 B up to 8 KB. As in LBFS [4], we imposed size limits to prevent

pathological cases; namely, a chunk cannot be smaller than $ecs/4$ and no larger than $ecs \times 2$.

The sender and receiver processes each ran in an Intel Core 2 Quad CPU machines with 8 GB RAM, connected by a 100 Mbps Ethernet network. All values presented next are an average of the results obtained on five runs of each experiment, preceded by one cold start run.

As for workloads, we consider three real workloads: *emacs*, *gcc* and *linux-src*. These workloads have been used to evaluate other state-of-the-art deduplication protocols (e.g. [8,10]). In each workload, R consists of a snapshot of a given set of files, whereas S corresponds to a subsequent snapshot of the same files. Initially, the receiver stores R. Each experiment consists of the sender sending S to the receiver. In *emacs*, the snapshots correspond to the source code trees of versions 20.1 and 20.7 (respectively) of the popular editor; in *gcc*, to versions 3.3.1 and 3.4.1 (respectively) of the compiler's source code tree; finally, *linux-src* denotes versions 2.4.22 and 2.4.26 (respectively) of the Linux kernel sources.

The following table summarizes each workload's characteristics that directly affect the behavior of HCs, as analyzed in Section 2.1.

	$|R|$	$|S|$	$P_{red}(S, R, ecs)$			
			128	512	2K	8K
emacs	43 MB	52 MB	57%	46%	33%	22%
kernel	149 MB	154 MB	90%	87%	80%	72%
gcc	135 MB	164 MB	61%	47%	33%	18%

Taking into account the above values and the lines plotted in Fig. 2, we chose to use the reference value of $k = 3.5$ bytes, which yielded negligible false candidate ratios. More precisely, only 0.17% false candidates were found with large chunks ($ecs = 8$ KB), which increased to a ratio of less 0.74% with small chunks ($ecs = 128$ B), for all workloads.

We start by studying the impact of HCs in transferred volume. Our results show that HCs are able to exchange substantially less bytes across the network than CBH. Fig. 3 illustrates such savings for the *linux-src* workload. As expected, the positive impact of HCs increases as we divide data into smaller chunks, therefore increasing the meta-data overhead associated with exchanging chunk hashes. HCs are able to significantly reduce meta-data volume by up to 33% in *linux-src*, relatively to CBH. In the case of *emacs* and *gcc*, which have lower redundancy levels than *linux-src*, meta-data savings rise to 62% and 64%, respectively. This confirms our predictions that HCs are more effective as redundancy decreases.

Most importantly, the savings on meta-data volume result in a tangible impact on the overall volume that HCs transfer (meta-data plus non-redundant data). In the case of *linux-src*, this amounts to an overall gain of 5.7% when compared to CBH, for $ecs = 2$ KB (the choice of ecs that minimizes the volume that CBH transfers). In *emacs* and *gcc*, we observe comparable overall volume gains (5.8% and 5.2%, respectively), again considering the ecs values in which CBH transfers less bytes.

It is worth noting that, in some cases, the meta-data savings of HCs can actually be traded for even greater data
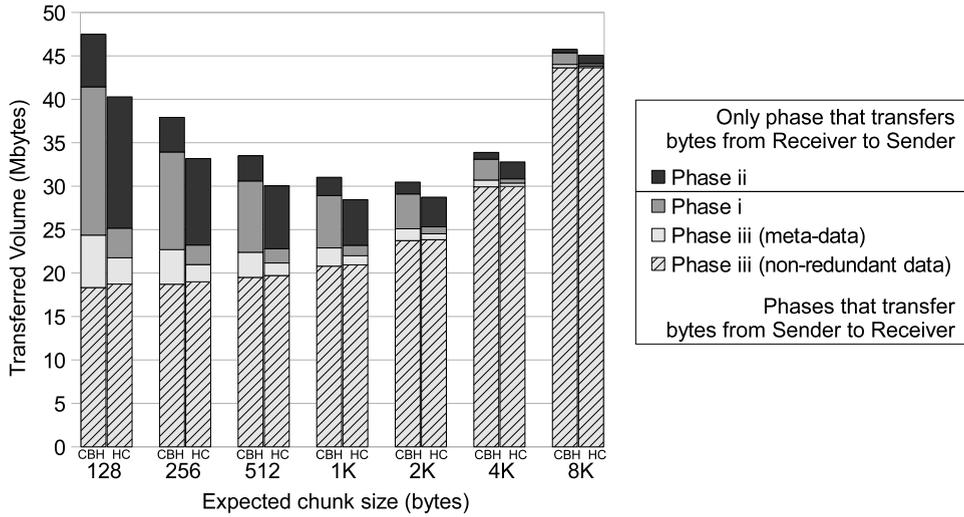
Fig. 3. Volume transferred at each phase of HC and CBH for different expected chunk sizes (*ecs*) in the *linux-src* workload.
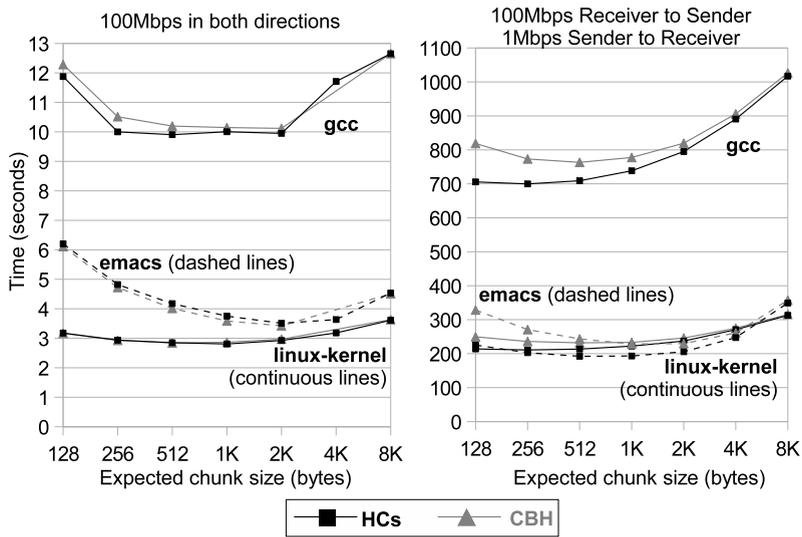
Fig. 4. Transfer times in symmetric (left-hand) and asymmetrical (right-hand) scenarios.

savings. For instance, considering the *linux-src* workload, by switching from an *ecs* of 2 KB to 1 KB, HCs are able to raise the overall volume gain to 6.6%, relatively to the best CBH option (2 KB).

Furthermore, if we restrict our focus to sender-to-receiver communication, the savings of HCs rise to 21.2%, 11.2% and 9.7% in *linux-src*, *emacs* and *gcc*, respectively. Again, this is consistent with our predictions.

We now turn our attention to the performance of HCs and CBH for each workload, which Fig. 4 presents. We start by studying the left-hand graph, corresponding to the baseline scenario where the sender and the receiver are connected by a symmetrical 100 Mbps link. If one considers the *ecs* values for which both solutions achieve best results in this scenario, we observe that the performance of HCs and CBH remains relatively close. For example, for *ecs* = 2 KB, HCs are 2.5% slower in *linux-src*, 1.5% faster in

emacs and 1.7% faster in *gcc*, when compared to CBH. This suggests that HCs, when compared to CBH, introduce no relevant local processing overhead.

When we shift to a scenario where the uplink has limited bandwidth, HCs start exhibiting substantial performance gains. The right-hand graph in Fig. 4 compares HC and CBH performance when we applied a 1 Mbps filter to the sender-to-receiver link. This recreates common broadband home Internet connections, with increasingly high downlink bandwidth and limited uplink bandwidth [2].

HCs attain considerable performance gains in this scenario. More precisely, HCs are 15.9% faster than CBH with *linux-src*, 9.0% with *emacs* and 8.3% with *gcc* (if we consider the best *ecs* choice for each solution in each workload). This is a natural consequence of the substantial gains in sender-to-receiver communication that HCs are able to attain.

4. Related work

Several research and industrial systems (e.g. [6,4,13–15]) employ CBH-based distributed deduplication. They either divide values into fixed-sized [6] or variable-sized chunks [4]. More recently, more intricate variants of CBH have been proposed that enhance precision and efficiency by relying on multi-resolution chunking schemes [8]. Examples include TAPER [8], Hierarchical Substring Caching [9], fingerdiff [10], JumboStore [11] and Wanax [16]. Although we describe HCs in the context of single-resolution, variable-sized CBH, our algorithm can be directly applied to improve all the above solutions.

If one regards R and S as chunk sets, CBH and HCs can be seen as probabilistic solutions to a problem that communication complexity literature calls *set intersection* [17]. The probabilistic communication complexity of set intersection is known to be $\Theta(n)$, where n is the length of the sets being compared [18]. HCs have equivalent communication complexity as long as one chooses a value of k for which the probability of false candidates is negligible.

5. Conclusions

The compare-by-hash (CBH) approach for distributed data deduplication has inherent precision and efficiency limits imposed by its meta-data overhead. We propose Hash Challenges (HCs), a novel technique that leverages existing CBH-based solutions by exchanging substantially less meta-data to ensure the same goal as CBH. We support our claims with a formal comparison of the network efficiency of our approach and CBH. Using real data from reference deduplication workloads, we show that hash challenges can save as much as 64% meta-data exchanged across the network, relatively to plain compare-by-hash. This implies reductions of up to 7% in overall transferred volume, resulting in performance gains of up to 16% in typical broadband connections.

Appendix A. Proof

We now prove the expression in Section 2.1 for the aggregate volume exchanged by HCs.

Phase i: Phase i will transfer k bytes per each one of the $|S|/cs$ chunks.

Phase ii: For each chunk c in S, the receiver can find candidates for two reasons: the receiver already stores c in R; or c' is a false candidate. The first case depends on probability $P_{red}(S, R, cs)$. Regarding the second case, let us first consider one chunk only, c, to send across the network. Assume, for simplicity, that the stored chunks at the receiver have the same average size as the chunks to send (cs). Hence, we know that the receiver stores approximately[2] $\frac{R}{cs}$ chunks in R. For each such chunk c', the

probability that it shares its k-byte hash prefix with c is $\frac{1}{2^{k\times 8}}$. Thus, the number of collisions with c is given by the mean of a binomial distribution with as many samples as chunks in R, each with the previous probability of success (i.e. collision), hence $\frac{R}{cs} \times \frac{1}{2^{k\times 8}}$. Since, for each chunk in S, any candidate (either true or false) results in sending a hash response of $N - k$ bytes plus its index back to the sender, we obtain the second term.

Phase iii: The sender starts by transferring an offset value per candidate it received, including false candidates (i.e., C). The sender then transfers the chunks that it did not find redundant. By definition of P_{red}, this comprises $[1 - P_{red}(S, R, cs)] \times \frac{S}{cs}$ chunks, each with an average size of cs.

References

[1] W. Hu, T. Yang, J.N. Matthews, The good, the bad and the ugly of consumer cloud storage, SIGOPS Oper. Syst. Rev. 44 (2010) 110–115.

[2] D. Belson, Akamai state of the Internet report, Q1 2011, Technical report, Akamai, August 2011.

[3] S. Ihm, K. Park, V.S. Pai, Towards understanding developing world traffic, in: Proceedings of the 4th ACM Workshop on Networked Systems for Developing Regions.

[4] A. Muthitacharoen, B. Chen, D. Mazieres, A low-bandwidth network file system, in: 8th ACM SOSP, 2001, pp. 174–187.

[5] N. Mandagere, P. Zhou, M.A. Smith, S. Uttamchandani, Demystifying data deduplication, in: Proceedings of the ACM/IFIP/USENIX Middleware Conference Companion, 2008, pp. 12–17.

[6] A. Trigdell, P. Mackerras, The rsync algorithm, Tech. rep., Australian National University, 1998.

[7] National Institute of Standards and Technology, FIPS PUB 180-1: Secure Hash Standard, NIST.

[8] N. Jain, M. Dahlin, R. Tewari, Taper: Tiered approach for eliminating redundancy in replica synchronization, in: USENIX FAST, 2005, p. 21.

[9] U. Irmak, T. Suel, Hierarchical substring caching for efficient content distribution to low-bandwidth clients, in: WWW, 2005, pp. 43–53.

[10] D. Bobbarjung, S. Jagannathan, C. Dubnicki, Improving duplicate elimination in storage systems, ACM Trans. Storage 2 (4) (2006) 424–448.

[11] K. Eshghi, M. Lillibridge, L. Wilcock, G. Belrose, R. Hawkes, Jumbo store: providing efficient incremental upload and versioning for a utility rendering service, in: USENIX FAST, 2007, p. 22.

[12] M.C. Chan, T.Y.C. Woo, Cache-based compaction: A new technique for optimizing web transfer, in: INFOCOM, 1999, pp. 117–125.

[13] L. Cox, B. Noble, Pastiche: Making backup cheap and easy, in: 5th OSDI, ACM, 2002, pp. 285–298.

[14] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, A. Perrig, T. Bressoud, Opportunistic use of content addressable storage for distributed file systems, in: USENIX Annual Technical Conference, 2003, pp. 127–140.

[15] S. Annapureddy, M.J. Freedman, D. Mazières, Shark: scaling file servers via cooperative caching, in: USENIX NSDI, 2005, pp. 129–142.

[16] S. Ihm, K. Park, V.S. Pai, Wide-area network acceleration for the developing world, in: USENIX Annual Technical Conference.

[17] T.G. Kurtz, U. Manber, A probabilistic distributed algorithm for set intersection and its analysis, Theor. Comput. Sci. 49 (1987) 267–282.

[18] B. Kalyanasundaram, G. Schnitger, The probabilistic communication complexity of set intersection, SIAM J. Discr. Math. 5 (4) (1992) 545–557.

---

[2] We make two conservative approximations here: that, for any c_1, c_2 in R, $c_1 \neq c_2$ and $c_1 \neq c$.