Efficient Locally Trackable Deduplication in Replicated Systems[⋆]

João Barreto and Paulo Ferreira

Distributed Systems Group - INESC-ID/Technical University of Lisbon
{joao.barreto,paulo.ferreira}@inesc-id.pt

Abstract. We propose a novel technique for distributed data deduplication in distributed storage systems. We combine version tracking with high-precision, local similarity detection techniques. When compared with the prominent techniques of delta encoding and compare-by-hash, our solution borrows most advantages that distinguish each such alternative. A thorough experimental evaluation, comparing a full-fledged implementation of our technique against popular systems based on delta encoding and compare-by-hash, confirms gains in performance and transferred volumes for a wide range of real workloads and scenarios.

Keywords: Data deduplication, data replication, distributed file systems, compare-by-hash, delta encoding.

1 Introduction

Many interesting and useful systems require transferring large sets of data across a network. Examples include network file systems, content delivery networks, software distribution mirroring systems, distributed backup systems, cooperative groupware systems, and many other state-based replicated systems [1]. Unfortunately, bandwidth remains a scarce, and/or costly in battery and price, resource for most networks [2], including the Internet and mobile networks.

Much recent work has proposed data deduplication techniques [3,4] for efficient transfer across the network, which may be combined with conventional techniques such as data compression [5] or caching [6]. Consider two distributed sites, a sender, S, and receiver, R. At some moment, each one locally stores a set of versions (of some objects, not necessarily the same set of objects at both sites), which we denote $\mathcal{V}_S$ and $\mathcal{V}_R$, respectively. If S wishes to send some of its local versions, $\mathcal{T}$ (where $\mathcal{T} \subseteq \mathcal{V}_S$), to R, some data chunks in $\mathcal{T}$ can be identical to chunks in $\mathcal{V}_R$. Data deduplication exploits such content redundancy as follows: when S determines that a chunk of data in some version in $\mathcal{T}$ is redundant, S avoids uploading the chunk and simply tells R where in $\mathcal{V}_R$ R can immediately obtain the redundant chunk.

The key challenge of data deduplication is in detecting which data is redundant across the versions to send and the versions the receiver site holds. The

[⋆] Funded by FCT grant PTDC/EIA/66589/2006.

difficulty of the problem greatly depends on the forms of chunk redundancy each approach tries to detect and exploit. We distinguish two such forms.

A first one arises from cases where newer versions borrow data chunks from older version(s) (of the same object, or of another object) that were locally available when the new versions were created. More precisely, consider some new version, v, created at S, that S wishes to send to R. If v shares a chunk with some older local version, v_{old} in $\mathcal{V}_S$, and R also happens to hold v_{old} (in $\mathcal{V}_R$), we say there exists *locally trackable redundancy*. In fact, in such conditions, if S (by some means) learns that v_{old} is common to both sites, S is able to detect the redundant chunk by merely comparing local versions (v and v_{old}).

Otherwise, we have *locally untrackable* redundancy. For instance, this is the case where two users, one at S and another at R, copy new identical data chunks from some common external source (e.g. a web page) and write it to new versions, v_S and v_R, that each user creates at her site, respectively. If S then decides to send v_S to R, clearly there exist redundant chunks in v_S that S can avoid transferring. However, in order to detect them, we must inevitably compare data chunks that are distributed across the two sites.

The prominent approach of *compare-by-hash*[1] [3,4,7,8,9,10] tries to detect both forms of redundancy by exchanging cryptographic hash values of the chunks to transfer, and comparing them with the hash values of the receiver's contents. Compare-by-hash complicates the data transfer protocol with additional round-trips (i), exchanged meta-data (ii) and hash look-ups (iii). These may not always compensate for the gains in transferred data volume; namely, if redundancy is low or none, or when, aiming for higher precision, one uses finer-granularity chunks [8,4,9]. Moreover, any known technique for improving the precision and efficiency of compare-by-hash [8,10] increases at least one of items (i) to (iii).

Earlier alternatives to compare-by-hash narrow the problem down to detecting locally trackable redundancy only. The most relevant example of *locally trackable deduplication* is delta-encoding [11,12,13,14,15]. Recent deduplication literature often regards such techniques as second-class citizens, leaving them out of the state-of-the-art that is considered when experimentally evaluating new deduplication solutions [3,4,7,8,9,10]. In part, two factors explain this. Firs, the inherent inability to exploit locally untrackable redundancy. Second, in the case of delta-encoding, the fact that, for some version, it can only detect redundancy with at most one other version (rather than across any set of versions, as in compare-by-hash) and the time-consuming algorithms involved [14].

This paper revisits locally trackable redundancy, proposing a novel combination of previous techniques. For a wide set of usage scenarios, we unify the advantages of both compare-by-hash and earlier locally trackable deduplication approaches. The key insight to our work is that detecting locally trackable redundancy exclusively is a much simpler problem, hence solvable with very efficient algorithms, than aiming for both forms of redundancy. It is easy to see that the problem now reduces to two main steps: (I) to determine the set of object versions that the sender and receiver site store in common; and (II) to determine

[1] We use the expression *compare-by-hash* for coherence with related literature, although it can have different meanings in other domains. By *compare-by-hash*, we mean *chunk-based deduplication through distributed hash exchange and comparison.*

which chunks of the versions to send are (locally trackable) redundant with such a common version set.

Since the versions that one must compare for redundancy detection (step II) now happen to exist locally at the sender site, we can now solve such a hard problem with local algorithms (instead of distributed algorithms that exchange large meta-data volumes over the network, such as hash values with compare-by-hash). No longer being constrained by network bandwidth, one can now: (i) perform a much more exhaustive analysis, thus detecting more locally trackable redundancy; (ii) and asynchronously pre-compute most of the algorithm ahead of transfer time. Furthermore, the network protocol is now a very simple one, (iii) with low meta-data overhead and (iv) incurring no additional round-trips: the receiver site simply needs to piggy-back some version tracking information in the request sent to the sender site (so that the latter performs step I).

For step I, we use very compact version tracking structures, called *knowledge vectors*. Most importantly, for step II, our approach can employ any local deduplication algorithm (e.g., [16,17,18]) for detecting locally trackable redundancy across *any* object and *any* version, with high precision and time efficiency. This is a significant distinction with regard to delta-encoding.

We have implemented our technique in the context of a novel distributed archival file system, called redFS. Using a full-fledged prototype of redFS for Linux, we have evaluated both single-writer/multiple-reader and multiple-writer usage scenarios based on real workloads, representative of a wide range of document types. By comparing redFS with popular state-of-the-art tools that rely on delta-encoding and compare-by-hash, we have confirmed our advantages over both approaches. Namely:

– redFS consistently transfers less (or, in few exceptions, comparable) bytes of data and meta-data than all evaluated solutions, obtaining reductions of up to 67% over relevant systems from both compare-by-hash and delta encoding approaches, namely LBFS [4], rsync [3] and svn [11].
– redFS transfers files considerably faster than all evaluated alternatives (or, exceptionally, comparable to the best alternative), accomplishing performance gains of more than 42% relatively to rsync, LBFS and svn, for networks of 11 Mbps and below. Except for svn, such gains even increase if we consider that file transfer starts after the local pre-computation steps of the redundancy algorithm have already completed. Furthermore, such speed-ups hold even for real workloads with relevant sources of locally untrackable redundancy, and assume reasonable log-space requirements.

The rest of the paper is organized as follows. Section 2 describes the system model. Section 3 then introduces our data deduplication protocol. Section 4 proposes two techniques that address the issue of efficient log storage. Section 5 describes the implementation of our protocol in redFS. Section 6 evaluates our solution, while Section 7 addresses related work. Finally, Section 8 draws conclusions.

2 System Model

We assume a replicated system where a number of distributed sites (denoted S, R, ...) can replicate a number of logical objects (such as files or databases). As we explain next, we impose little restrictions on the replication protocol, hence making our solution widely applicable. Each site has a pre-assigned, well-known unique identifier. For simplicity of presentation, and without loss of generality, we assume all sites are trusted, as well as communication.

When a local applications writes to a local replica, it creates a new value of the object, called a *version*. Replicas maintain a finite *version logs*, with their latest version histories. The oldest logged versions may be pruned at anytime.

To simplify presentation, we assume that no write contention exists, hence no version conflicts exist. Nevertheless, our solution can be transparently combined with commitment protocols [1] to handle concurrent/conflicting versions.

We divide the object space into disjoint sets of objects, called *r-units*. We assume the individual r-unit to be the minimum replication grain in the underlying system; possibly, a r-unit may comprise a single file. Each site may replicate any arbitrary set of r-units. A site replicating a r-unit maintains replicas for all objects of that r-unit. As we explain later, larger r-units allow sites to maintain, exchange and analyze less bytes and perform faster data deduplication; while smaller r-units enable finer granularity in the choice of which objects to replicate. The problem of optimal partitioning of the object space into r-units is out of the scope of this paper. For simplicity of presentation, and again without loss of generality, the set of sites replicating each r-unit is assumed to be static.

Each logged version is identified by the identifier of the site where the version was created (denoted $v.creator$) and a sequence number (denoted $v.sn$). Hence, even if two objects happen to be created at different sites with the same textual name, their actual identifier ($\langle v.creator, v.sn \rangle$) is unique. For a given r-unit, the sequence number monotonically increases among the versions created by site $v.s$ at any object belonging to that r-unit. For example, given r-unit $u = \{a, b\}$, if site S creates a new version of object a, then another version of object b, the first version will be identified by $\langle S, 1 \rangle$, the second one by $\langle S, 2 \rangle$, and so on.

According to some arbitrary replication protocol, sites replicating common r-units exchange new versions to bring each other up-to-date. Without loss of generality, such a step occurs in unidirectional pair-wise synchronization sessions, in which a sender site, S, sends a set of versions, $\mathcal{T}$, to a receiver site, R.

3 Data Deduplication Protocol

We now describe the data deduplication protocol, which complements the underlying log-based replication protocol. Our solution may be seen as a combination of a space-efficient version tracking and local similarity detection algorithms.

We follow such a structure in the following sections. Recall from Section 1 that the generic approach to locally trackable deduplication consists of 2 steps: (I) to determine the set of object versions that the sender and receiver site store in common; and (II) to determine which chunks of the versions to send are (locally trackable) redundant with such a common version set. We describe our novel solution to each such step in Sections 3.1 and 3.2, respectively. Section 3.3 then combines the two steps in a complete deduplication protocol.

3.1 Step I: Version Tracking

Given a pair of sites, S and R, that are about to synchronize, the goal of Step I is to make S infer the the set of common versions both sites currently share. We denote such a set as $\mathcal{C}$. One naive solution would be to maintain and exchange lists of version identifiers for each r-unit they replicate. Of course, the inherent space requirements, as well as network transfer and look-up times, would easily become prohibitive for systems with reasonably large object/version sets.

Instead, we rely on a pair of vectors, each with one counter per site replicating the r-unit, which we call *knowledge vectors*. Each site, S, maintains, along with each r-unit, u, that S replicates, two knowledge vectors, denoted $K^i_S(u)$ and $K^f_S(u)$. From $K^i_S(u)$ and $K^f_S(u)$, we can infer that S currently stores any version, v, of any object in u such that $K^i_S(u) \leq v.sn \leq K^f_S(u)$.

Such a pair of vectors can only represent a version set in which every version created by the same site i (for all i) have sequence numbers that can be ordered consecutively without any gap. When, for each r-unit a site replicates, the set of versions it stores satisfies the previous condition, knowledge vectors constitute a very space- and time-efficient representation. To simplify presentation, for the moment we assume that sites receive and prune versions in sequential number order, with no gaps. Under this assumption, we can precisely represent a site's version set by pairs of knowledge vectors, one for each r-unit the site replicates.

This assumption is not artificial, as most log-based replication protocols systems either ensure it (e.g. Bayou [19]), or may easily be adapted to accomplish it (e.g. Coda [20]). More generally, it is trivial to show that any system that guarantees the widely-considered *prefix property* [19] satisfies such an assumption. Section 4 then addresses cases where version gaps can occur.

Given some site, S, and a r-unit that site replicates, S maintains $K^i_S(u)$ and $K^f_S(u)$ as follows. Both vectors start as zero vectors. As S accepts a new write request, creating a new version v, it updates its vectors to reflect the new version: i.e. $K^f_S(u)[S] \leftarrow v.c$ (and, if $K^i_S(u)[S] = 0$, then $K^i_S(u)[S] \leftarrow v.c$). Furthermore, if S deletes a given version, v, it sets $K^i_S(u)[S] \leftarrow v.c + 1$ (recall that, for now, we assume that version deletions occurs in sequential number order). Besides versions that the local site creates and deletes, S's version set will also evolve as S receives a new version, v, from some remote site. In this case, S sets $K^f_S(u)[S] \leftarrow v.c$ (again, assuming the ordered propagation with no gaps).

When the receiver site, R, sends a first message requesting for synchronization to start from a sender site, S, R *piggy-backs* the knowledge vectors of each r-unit that R replicates. Thus, imposing no extra round-trips on the underlying synchronization protocol, S can infer which versions are common to both sites.

While such knowledge vectors tell S which versions R held *right before* synchronization started, at some point *during* synchronization R will already have received other versions (during the current synchronization session from S). Evidently, set $\mathcal{C}$ also includes the latter versions. In order to keep track of them, S maintains, for each r-unit, u, involved in the current (ongoing) synchronization session, a pair of knowledge vectors, denoted knowledge vectors $t_i(u)$ and $t_f(u)$. Both vectors start null (when synchronization starts) and grow to represent each version (of the corresponding r-unit) that S sends.

From the copy S has received of R's knowledge vectors, as well from $t_i(u)$ and $t_f(u)$ (for each r-unit involved in the synchronization session), S can easily determine whether a given version that S stores, v_S, is also stored at R (i.e. whether $v_S \in \mathcal{C}$) by testing if any of the following conditions hold:

$$K_R^i(v_S.runit)[v_S.creator] \leq v_S.sn \leq K_R^f(v_S.runit)[v_S.creator]$$
$$\text{or}$$
$$t_i(v_S.runit)[v_S.creator] \leq v_S.sn \leq t_f(v_S.runit)[v_S.creator].$$

Hence, we can determine whether $v_S \in \mathcal{C}$ in constant time, with few operations. The first condition means R already stored v_S before synchronization started, while the second one means that R has received v_S in the meantime.

3.2 Step II: Local Chunk Redundancy Detection

Given a set of versions to transfer, $\mathcal{T}$, from site S and R, and a set of common versions between S and R, $\mathcal{C}$, Step II determines which chunks of $\mathcal{T}$ are redundant across $\mathcal{C}$. We solve step II very efficiently by observing that most of it needs not be computed synchronously while the synchronization session is taking place.

We divide step II into 2 sub-steps. A first sub-step is the most computationally intensive. It detects redundancy across *every* version that S stores, writing its findings to *redundancy vertices* and per-version *redundancy vertex reference lists,* both maintained locally. This sub-step is independent of any synchronization session. Hence, it can be computed ahead of synchronization time as a background thread that updates the local redundancy vertices when S is idle.

When synchronization starts, only the second sub-step needs to run: determining, for the ongoing session, which chunks of $\mathcal{T}$ are redundant across $\mathcal{C}$. This sub-step takes advantage of the pre-computed data structures, being able to return a result by a simple query to such data structures.

The next sections describe each sub-step in detail.

Pre-Computing Chunk Redundancy Data Structures. Conceptually, we divide the contents of all the versions a site stores into disjoint contiguous portions, called *chunks.* We determine locally trackable redundancy relationships among the entire set of chunks a site stores by running some (local) redundancy detection algorithm. If such an algorithm finds that two chunks have identical contents, we designate them as *redundant across the set of versions.* A chunk with no redundant chunks is called *literal.*

In a typical configuration, the redundancy detection algorithm runs as a low-priority thread, which starts from time to time. At each run, it analyzes the set of local versions that, relatively to the last run of the algorithm, are new, comparing their chunks with the chunks of the remaining versions.

The resulting redundancy relationships are maintained in local data structures called *redundancy vertices.* Each redundancy vertex either represents a literal chunk or a set of redundant chunks (sharing identical contents). A redundancy vertex is simply a list of *[version identifier, byte offset, chunk size]* tuples, each pointing to the effective contents of a chunk within the local versions.

As we explain later, redundancy vertices should ideally reside in main memory, for the sake of synchronization performance. However, for sufficiently large data repositories, that may not be possible due to the fine chunk granularity our solution is designed for (e.g., 128 bytes). For example, in a file system with an average chunk size of 128 bytes, supporting up to 4G file versions, each up to 1 TB, would require 9-byte chunk pointers at redundancy vertices (one pointer per stored chunk). Thus, redundancy vertices would introduce a space overhead of 7%; e.g., 60 GB of version contents would imply 4-GB of redundancy vertices. If such space requirements exceed the available primary memory, we can store the redundancy vertices in secondary memory and maintain the elements that are more likely to be consulted in upcoming synchronization sessions in a cache in main memory (for instance, using a *least-recently-used* substitution policy). A detailed evaluation of this option is left for future work.

Any version that the redundancy detection algorithm has already analyzed has each of its chunks referenced by exactly one redundancy vertex. Along with each such version, the redundancy detection algorithm stores an ordered list of references to the redundancy vertices corresponding to each chunk in the version.

Our contribution is not tied to any particular redundancy detection algorithm, and we can transparently use any algorithm that works for local deduplication. Namely, approaches such as the fixed-size sliding block method of rsync [3], diff-encoding [14], similarity-based deduplication [18], among others [16].

Nevertheless, to better illustrate our solution, and without loss of generality, we hereafter consider one of such possible alternatives: a local variant of LBFS's distributed algorithm [4]. This algorithm is able to detect redundancy spanning across any versions, possibly from distinct objects or r-units.

Very succinctly, the algorithm works as follows. We maintain a *chunk hash table*, whose entries comprise a pointer to a redundancy vertex and the hash value of contents of the corresponding chunk(s). For each new version (either locally created or received from a remote site), we divide it into chunks (using LBFS's sliding window fingerprinting scheme [4]) and look their hash values up in the chunk hash table. If we find a match, then there exists at least one other identical chunk. In this case, we add a pointer to the new chunk to the existing redundancy vertex. Otherwise, the new chunk is literal, thus we create a new single-chunk redundancy vertex pointing to the chunk, and add a new entry to the chunk hash table (referencing the redundancy vertex and the chunk's hash value). Once we finish processing each version, we trivially construct the corresponding redundancy vertex reference list and store it along with the version.

Similarly to the previous discussion concerning redundancy vertices, maintaining an in-main-memory chunk hash table that covers every local fine-grained chunk can be impossible. Solving this problem is out of the scope of this paper. Possible directions include storing the chunk hash table in secondary memory and a cache in main memory, or resorting to similarity-based schemes [18].

Determining Redundant Chunks Across $\mathcal{T}$ and $\mathcal{C}$. Using the previously maintained data structures (redundancy vertices and redundancy vertex reference lists) as input, the actual step of determining which chunks to transfer are actually redundant across both sites is reduced to simple queries to such information.

More precisely, for each version v that site S is about to transfer (i.e. $v \in \mathcal{T}$), we iterate over its redundancy vertex reference list and, for each reference, we (1) read the corresponding redundancy vertex; and (2) iterate over the redundancy vertex's chunk pointer list until we find the first chunk of a version $v_r \in \mathcal{C}$ (using the expression from Section 3.1). If the latter chunk is found, we can just send a reference to where, among the versions R stores, R can obtain the redundant chunk. More precisely, the following *remote chunk reference* is sufficient to univocally identify such a location: *[v_r.id, offset, chunk size]*.

Since we consider fine-grained chunks, it is frequent to find consecutive remote chunk references to contiguous (and consecutive) chunks in some version in $\mathcal{C}$. We optimize these cases by coalescing the consecutive remote chunk references into a single reference to a larger chunk comprising all the consecutive smaller chunks. This simple optimization has the crucial importance of dissociating the local redundancy detection granularity from the effective remote chunk reference volume that we send over the network.

As it is shown elsewhere [21], reference coalescing ensures that, by decreasing the average chunk size, either: (i) we transfer the same remote chunk reference volume (if no additional redundancy is detected); or (ii) the increase in remote chunk reference volume is compensated by the decrease in transferred data volume (if smaller chunks unveil more redundancy).

The algorithm above implies $O(d)$ verifications for membership in $\mathcal{C}$ using the condition from Section 3.1, where d is the average cardinality of each set of redundant chunks in a redundancy vertex. In contrast, LBFS's approach requires look-ups to the receiver site's chunk hash table that involve $O(log(n))$ hash comparisons, where n is the total number of local chunks. As, for most workloads, $d << logn$, the local computation phase of our protocol is substantially faster than the one of LBFS (an observation we confirm in Section 6).

3.3 Putting It All Together

After running the previous steps, site S can finally transfer each version $v \in \mathcal{T}$ to R. The actual transfer of each such version involves sending three components. Firstly, v's identifier and size. Secondly, an array of remote chunk references to redundant chunks. We send remote chunk references in the order by which the corresponding chunks appear in v. Finally, we send the contents of every literal chunk of v, again in their order of appearance within the version.

Upon reception of the above components, R starts reconstructing v by copying the redundant contents from the locations (among R's versions) that the remote chunk references point to. The gaps are then filled with the literal contents that R has received.

4 Log Storage and Maintenance

The effectiveness of our data deduplication solution depends on the ability of each site to maintain long-term version logs. To substantially increase log efficiency, we can resort to two compression schemes.

A first scheme is called *redundancy compression*. The principle behind redundancy compression is the same as in local deduplication solutions [17]: whenever two or more versions share a common chunk, we store the chunk only once and suppress the redundant copies. We achieve this by storing only the first chunk referenced at each redundancy vertex. If there are additional chunks at that redundancy vertex, we set an *absent* flag at the redundancy vertex reference list of the versions such chunks belong to. This flag means that, to read the chunk's contents, we must follow the first chunk pointer in the redundancy vertex.

When redundancy compression is insufficient, we inevitably need to erase both the redundant and literal chunks of one or more versions, thus losing their contents. Still, we can retains the version's redundancy vertex reference list, which constitutes a *footprint* of the redundancy relationships of the erased version with other versions. The footprints of versions that have been erased at the sender site can still be (artificially) considered when determining the set of common versions, $\mathcal{C}$, during synchronization. In fact, if S determines that some version, v, for which S only holds its footprint (since it has already discarded its contents) is (fully) stored at the receiver site, R, then S considers v to be in $\mathcal{C}$. Hence, if S determines that some chunk to transfer is redundant with v (which S can determine by simply inspecting v's footprint), then S can exploit such redundancy and send a remote chunk reference to version v that R stores. Notice that the fact that S has already erased the contents of v is irrelevant as long as R still stores them. Incorporating footprints into the protocol described so far is straightforward; for space limitations, we describe it elsewhere [21].

Eventually, completely erasing both contents and footprint of a version will also occur, once space limitations impose it. In this case, care has to be taken in order to update the knowledge vectors of the r-unit in which the object being pruned is included. More precisely, such vectors need to be changed in order to reflect the smaller version set of the local site.

However, the completely erased version can cause a gap in the version set that was previously denoted by the knowledge vectors. For instance, if $K^i_S(u)[S] = 1$ and $K^f_S(u)[S] = 10$ and the user decides to delete version $\langle S, 3\rangle$. In this case, we can no longer use the knowledge vectors to represent the version set. Instead, we set such vectors to represent a the *subset with highest identifiers* that has no gaps. Recalling the previous example, we would set $K^i_S(u)[S] \leftarrow 4$. The choice for the subset with the highest identifiers follows the heuristic that since versions with higher identifiers are probably most recent, then they are more likely to be synchronized to other sites. Therefore, it is more valuable for our deduplication solution to keep track of such versions than those with lower identifiers.

Similarly to complete erasure of a version, gaps can also happen when a site receives new versions from a remote site. In this case, the same solution is taken.

5 Implementation

We have instantiated our solution in a full-fledged distributed archival file system, called redFS.[2] redFS supports distributed collaboration through file

[2] Source code available at: `http://www.gsd.inesc-id.pt/~jpbarreto/redFS.zip`

sharing. Applications can create shared files and directories, and read from/write to them in an optimistic fashion. redFS implements on-close version creation [22].

redFS runs on top of the FUSE [23] kernel module, version 2.6.3, in Linux. For synchronization, redFS runs a variant of Bayou's update propagation protocol [19], complemented with our data deduplication scheme. We use SUN RPC over UDP/IP, resorting to TCP/IP sockets to transfer literal contents.

The unit of storage is what we call a *segment*. Each segment has a unique version identifier and is stored as an individual file in a native file system. Every directory and file version has its data stored in a segment.

For directories, we adopt a similar solution as in the Elephant file system [22]. A directory is stored in a single segment, identified by the directory version, and comprising a directory entry for each object in the directory. Changes to the directory (e.g., objects added, removed, or attributes changed) result in appending a new entry to the directory segment and setting an `active flag` (see [22]) of the previous entry corresponding to the affected object to false.

Each directory references a set of files/directories by storing the corresponding *file identifiers*. A file identifier of a given file/directory consists of the version identifier of the initial version of the file/directory. To open a file we need to map the file identifier to the identifier of its current version. redFS maintains a version information table, indexed by file identifier, which contains, for each file: version identifier, size, compression mode (*plain*, *redundancy* or *footprint-only*), and the segment identifier of the file log (if any), among other flags. If a file has only a single version, it has no log segment. This is an important optimization since many files of a file system are never changed. Otherwise, the log consists of a list of version information entries, similar to the ones in the version information table, providing access to archived versions.

A similarity detector thread runs asynchronously, and an explicit shell command starts each run. Currently, we maintain redundancy vertices and the chunk hash table exclusively in main-memory. Redundancy vertex reference lists are stored at the head of the corresponding versions.

6 Evaluation

Assuming a synchronization session from site S to site R, which transfers a set of versions, $\mathcal{T}$, an efficient deduplication approach should minimize the following metrics:

1. *Transferred volume*, the amount of bytes (including both data and meta-data) transferred across the network in order to transfer $\mathcal{T}$;
2. *Full transfer time*, the time it takes for S to, immediately after obtaining $\mathcal{T}$, transfer the versions in $\mathcal{T}$ to R.
3. *Foreground transfer time*, the time it takes for S to transfer $\mathcal{T}$ to R, assuming S has already had, in background, the opportunity to perform some local pre-computation phase.
4. *Space Requirements*, the amount of memory that both sites need to maintain.

We now evaluate redFS under different workloads and network settings, both in single-writer and multiple-writer scenarios. We compare redFS with relevant alternatives employing compare-by-hash and delta-encoding, and plain file transfer. Our evaluation tries to answer 2 questions. Firstly: *how does redFS compare to its alternatives in terms of transferred volume, full transfer time and foreground transfer time, assuming unbounded space resources?* Secondly: *how does redFS's efficiency degrade as the available (bounded) space resources decrease?*

For space limitations, the following sections discuss the most relevant results only. An exhaustive evaluation can be found in [21].

6.1 Experimental Setting

The experiments consider two sites replicating files from real workloads. As new versions are produced at either one or both sites, they synchronize their local replicas. To simplify our analysis, we start by assuming that both sites replicate a common r-unit only and have unbounded logs. Later in the section, we lift these restrictions, and extend our analysis to multiple r-units and bounded logs.

A first site, S, runs Ubuntu 6.06 (kernel version 2.6.15) on a Pentium 4 3GHz processor with 1GB of RAM. The second site, R, runs Debian 4.0 (kernel version 2.6.18) on an Athlon 64 processor 3200+ with 2 GB of RAM. All experiments were run during periods of negligible load from other processes running at each machine (approximately $< 1\%$ CPU usage by other applications). A 100 Mbps full-duplex Fast Ethernet LAN interconnected both sites. For more complete results, we use a class based queue (CBQ) to emulate relevant network technologies, namely IEEE 802.11g (54 Mbps) and 802.11b (11 Mbps) wireless LANs, Bluetooth 2.0 personal area networks (3 Mbps). Performance measurements were taken with the `time` command of Linux, with millisecond precision. The presented results are averages of 3 executions of the each experiment.

We evaluate a representative set of solutions, covering all relevant actual approaches for network traffic deduplication:

– ***redFS*** with different expected chunk sizes (128 bytes, 2 KB, 8 KB).[3]
– Solutions based on compare-by-hash, namely: ***lbfs***, our implementation of LBFS's original protocol [4], in the same modes as for redFS (we do not directly evaluate LBFS because no public stand-alone implementation of the original solution was available); ***rsync*** [3], version 2.6.3 of the popular compare-by-hash Linux tool, using the default fixed chunk size, 700 bytes; and ***TAPER***, using the published results [8] (no prototype nor full source code was publicly available).
– *svn* [3], version 1.4.6 of the popular distributed version control system, which relies on delta encoding for committing versions to the server, using its most efficient server (*svnserve*).
– Finally, as a base reference, we also evaluate ***plain*** remote file transfer.

The results presented herein were obtained with data compression [5] disabled. Not considering data compression simplifies our analysis, as it excludes any measurement noise caused by different data compression algorithms and options used

[3] The choice of chunk sizes is driven by the observations that: (i) lower chunk sizes than 128 bytes do not compensate the meta-data overhead of remote chunk references, and (ii) higher chunk sizes than 8 KB yielded no advantage.

by each evaluated system.[4] The exception is *svn*, in which data compression is hard-coded. For a fair comparison, we present measurements for a hypothetical variant of *svn* with no data compression, extrapolated by *svn*'s performance relatively to the one of *rsync* with compression on.

We replay replica updates with workloads taken from real world collaborative situations, ranging from more than 100 MB up to more than 500 MB, and from more than 850 files up to more than 42,000 files.

Single-Writer, Multiple-Reader Scenarios. A first group of workloads consists of workloads that relevant compare-by-hash work uses as their evaluation basis; namely, LBFS's [4] and TAPER's [8] original papers consider either such workloads or very similar ones. These workloads reflect a single-writer, multiple-reader scenario. It is easy to see that, by definition, all redundancy in this scenarios is locally trackable (assuming sufficiently large version logs).

Site S acts as a writer that sequentially produces two sets of versions in the r-unit. Site R is a reader, which synchronizes after each version set is ready.

We test workloads of different categories in such a scenario:

Software development sources. These workloads are representative of collaborative code development scenarios. They include, for different real-world open-source projects, two consecutive versions of their source code releases. We have selected the source trees of recent versions of the gcc compiler (versions 3.3.1 and 3.4.1), the emacs editor (20.1 and 20.7), and the Linux kernel (2.4.22 and 2.4.26). Hereafter we call such workloads `gcc`, `emacs` and `linux-source`, respectively. Nearly all redundancy in these workloads ($> 95\%$) is found between pairs of files with the same name (each from each version). The choice of projects and versions is the same as adopted for the evaluation of TAPER [8]; this allows us to compare our results with theirs.

Operating system executable binaries. One workload, `usrbin`, considers binary files, which have very different characteristics when compared to the previous text-based files (namely, in data compressibility and cross-file and cross-version redundancy). It includes the full contents of the `/usr/bin` directory trees of typical installations of the Ubuntu 6.06 32-bit and 7.10 64-bit Linux distributions, which include most of the executable code binaries that are bundled with a Linux installation. All redundancy is between files with the same name.

Multiple-Writer Scenarios. A second group of workloads consider multiple-writer scenarios, where actual concurrency (hence, locally untrackable redundancy) can finally occur. In this case, two writer sites, start with a common initial version set. Each writer site then independently modifies its local replicas. Finally, both sites synchronize their divergent replicas.

We consider two workloads, obtained from real collaborative document editing scenarios. Both result from real data from two undergraduate courses of Technical University Lisbon. Their data spans across a full semester.

A first workload, called `course-docs`, consists of snapshots of two CVS repositories shared by lecturers of each course to keep pedagogical material (e.g. tutorials, lecture slides, code examples and project specifications) and

[4] We confirm in [21] that enabling data compression yields equivalent conclusions.

private documents (e.g. individual student evaluation notes and management documents).

The initial version of the workload includes the contents of both repositories at the beginning of the semester. The type of files varies significantly, ranging from text files such as Java code files or html files, to binary files such as pdf documents, Java libraries, Microsoft Word documents and Microsoft Excel worksheets. Significant data was copied across different files: only 30% redundancy is found between versions with identical file names. Both courses were tightly coordinated, as their students are partially evaluated based on a large code project that is common to both courses. Consequently, lecturers of both courses carried an eminently collaboratively activity, involving regular meetings. Inevitably, both repositories had regular locally untrackable interactions, such as frequent email exchanges, and weekly meetings.

A second workload, `student-projects`, captures the collaborative work among students of both courses. Teams of 9 students were given the assignment of developing a code project in Java, which they were demanded to develop on a CVS repository. To ease their coding effort, they were provided with a bundle of auxiliary Java libraries and illustrative code examples. Most teams relied on this bundle as a basis from which they started developing their final project.

The project lasted for 3 months. The initial version set is a snapshot of the initial library/code bundle that was made available to every student. We then randomly selected two final projects from the CVS repositories, which constitute a divergent version set at each site. The snapshots consist mainly of binary Java libraries, Java code files, as well as Microsoft Word and pdf documents. Again, this workload had sources of locally untrackable redundancy; e.g. code examples that the students incorporated into their project were provided to both teams through the courses' web sites, hence locally untrackable.

6.2 Results

This section is organized as follows. We start by focusing on transferred volume, full and foreground transfer times. We start by considering the single-writer, multiple-reader scenario, where no locally untrackable redundancy can arise, and assuming infinite logs. We then depart to the expectedly more challenging multiple-writer scenario, where locally untrackable does occur. Finally, we analyze space requirements associated with finite logs of increasing depths.

Single-Write, Multiple Reader Scenarios. In a first experiment, we ran all workloads of the single-writer case, plus single-writer variants of `course-docs` and `student-projects` (considering only one of the two divergent versions).

On average over *all* workloads, the volume that redFS transfers across the network during synchronization is substantially lower than *rsync* and *lbfs* (on average over all workloads, both *rsync* and *lbfs-128* transfer 35% more bytes than *redFS-128*). TAPER's intricate solution transfers comparable volume to redFS (TAPER transfers 0.29% less on average).

With respect to delta encoding tools, since ***svn*** does not output transferred volumes, we evaluate the ***xdelta*** [24] local delta-encoding tool for such a measurement. redFS is, in general, more efficient (9% lower volume on average over all workloads). However, for workloads with a sufficiently stable directory/file tree and where in-object redundancy dominates cross-object redundancy (which is the case of the `gcc` workload), delta encoding can substantially outperform redFS. Hence, an interesting direction for future improvement is to complement redFS's similarity detection and reference encoding with a variant based on delta encoding. Such a hybrid approach is clearly compatible with our solution.

Figure 1 illustrates transferred volumes for a workload where most redundancy is across consecutive versions of the same file (gcc), and for a workload where cross-file redundancy is significant (course-docs).

Figure 1 includes *redFS-128 NCL*, a variant where we disable reference coalescing. It illustrates how crucial the optimization is to redFS when using fine-grained chunks. Coalescing references allows redFS-128 to send 8x and 14x less remote chunk references in `gcc` and `course-docs`, respectively. Overall, *redFS-128 NCL* entails 18% and 11% overheads in transferred volume, respectively.

More important than transferred volume is to analyze the actual performance of redFS relatively the other solutions, namely in terms of their full and foreground transfer times. Figure 2 presents such measurements for three very distinct workloads: `emacs`, a highly-redundant workload (more than 54% redundant chunks), where most redundancy occurs between consecutive versions of the same file; `usrbin`, a workload exhibiting less than 30% redundant chunks; and `course-docs`, with relatively high cross-file redundancy.

In the case of redFS, we depict the full and foreground transfer times with 128 bytes expected chunk size (resp. labeled `redFS-128-fullsynch` and `redFS-128`), as such variant exhibited best performance among the others. Regarding ***svn*** and ***rsync***, we can only obtain their full transfer time, as their implementations only consider a full synchronization option (resp. `svn-fullsynch` and `rsync-fullsynch`). Just for curiosity, we also depict ***svn***'s original full transfer time, with data compression on. Regarding our implementation of lbfs, for presentation simplicity, we depict its foreground transfer times only, and consider the highest performance variant, lbfs-8KB. Finally, in the case of plain transfer, its full and foreground times are, by definition, the same.

Perhaps the strongest conclusion from this analysis is that, considering all workloads, redFS-128's average full transfer times are lower than the same

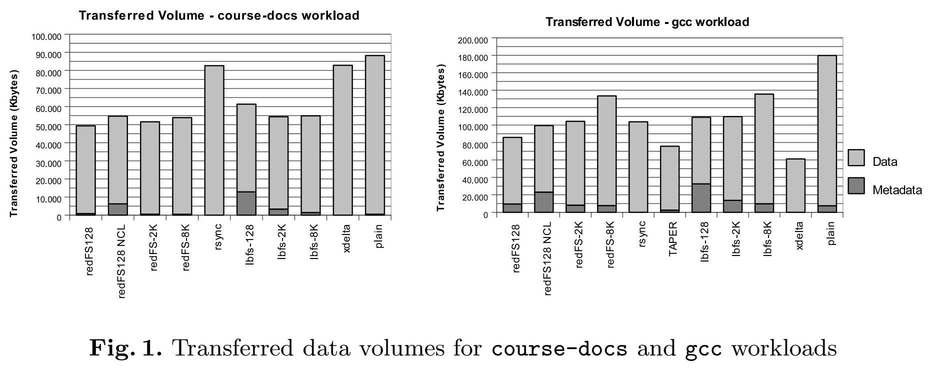

Fig. 1. Transferred data volumes for `course-docs` and `gcc` workloads

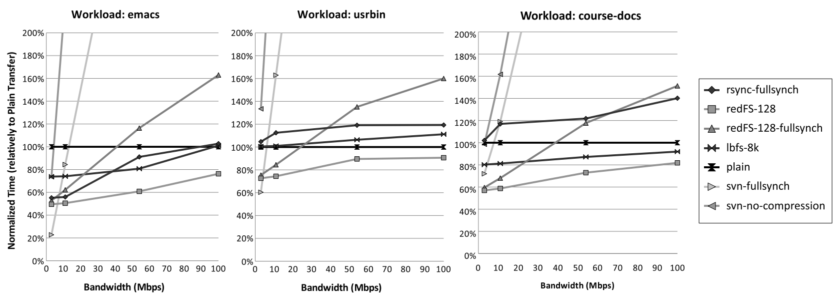

Fig. 2. Transfer times for different workloads, for varying bandwidths

measurement of *every* other evaluated solution (with data compression off), for bandwidths of 11 Mbps and below. Furthermore, this conclusion is true for workloads with very distinct characteristics, which Figure 2 shows. As we consider higher bandwidths, however, the time penalty of redFS-128's local chunk detection algorithm becomes increasingly decisive.

However, as Section 3.2 discusses, our technique is mainly intended to have the local chunk detection phase running in background. In typical cases, when the user initiates synchronization with another site, such a background algorithm will already have completed and, therefore, the only delay that the user will notice is the foreground transfer time – therefore, the most important metric.

Relatively to lbfs-8KB, redFS-128's foreground transfer time is consistently lower (from 5% with 100 Mbps to 34% lower with 3 Mbps). This is easily explained by lbfs's exchange of hash values across the network, as well by its lower precision in detecting redundancy (as analyzed previously). Naturally, the impact of such drawbacks become more dominant as bandwidth drops.

Unfortunately, we cannot precisely compare redFS's foreground transfer time with ***rsync***'s. However, the portion of ***rsync***'s algorithm that one can precompute is significantly smaller than in the case of redFS (most notably, ***rsync*** must exchange hash values of the versions to transfer in foreground). Hence, we can safely infer that, for the workloads where redFS's full transfer time is lower than ***rsync***'s, redFS's foreground transfer times will also outperform ***rsync***'s.

Since TAPER's source code is not fully available, we could not consider it in the present performance analysis. Nevertheless, our observations are that (i) the data volumes transferred by redFS and TAPER are comparable, as well as that (ii) redFS's single-round-trip protocol is expectedly lighter than TAPER's intricate 3-round-trip protocol. These strongly suggest that redFS would be faster than TAPER in the evaluated workloads.

Moreover, concerning ***svn***'s delta-encoding solution, clearly its local delta encoding delays dominate ***svn***'s full transfer times. Nevertheless, for the specific workloads in which delta encoding yielded lower transferred volumes, we expect ***svn*** to have lower foreground transfer times than redFS.

It is worth noting that, with the low redundancy `usrbin` workload, redFS-128 was the only solution to achieve an actual speed-up in foreground transfer time, an evidence of redFS's low protocol overhead.

Finally, we repeated the same experiments with multiple r-units in common between both sites. Theoretically, we know that, with r common r-units, (i) the receiver site sends r knowledge vectors in the message requesting synchronization; and (ii) for each local chunk that is redundant with a chunk to send, the sender site performs r verifications (in the worst case) of the conditions in Section 3.1. The impact of (i) is negligible for the considered workload sizes, even for the lowest bandwidths. Concerning factor (ii), we observe that its impact is limited (less than 10% higher foreground transfer time) as long as we consider less than 1000 common r-units, even with highly redundant workloads and high bandwidths. For instance, with `emacs` (54% redundancy detected) and 100 Mbps, ***redFS-128***'s foreground transfer time drops linearly as the number of common r-units grows, reaching the 10% degradation limit at 2000 r-units.

Multiple-Writer Scenarios. In contrast to the previous single-write scenarios, multiple-writer scenarios, such as those arising from replicated systems supporting distributed collaborative work, incur locally untrackable redundancy. In order to assert whether redFS is still advantageous in such scenarios, we need to quantify locally untrackable redundancy. For that, we consider the two concurrent workloads, `course-docs` and `student-projects`.

The methodology for quantifying locally untrackable redundancy is to measure how much data volume *redFS-128* would be able to detect as (locally trackable) redundant, versus the data volume that *lbfs-128* would detect as redundant (both locally trackable and untrackable), and then to subtract both values.

Perhaps surprisingly, the redundancy that results from locally untrackable data exchanges during several months is almost insignificant when compared to locally trackable redundancy. In the `course-docs` workload, *lbfs-128* detects just 1,01% more (locally untrackable) redundant contents than redFS. The `student-projects` workload exhibits more locally untrackable redundancy, but still at relatively insignificant levels: *lbfs-128* detects 4,37% more redundant data.

Our results confirm that locally untrackable redundancy does occur in concurrent scenarios. But, most importantly, they show that locally trackable redundancy strongly dominates locally untrackable redundancy. Hence, the advantages of redFS over other state-of-the-art alternatives that the previous sections exposed are also effective in an important class of multi-writer scenarios.

We then evaluated redFS, *rsync* and lbfs when synchronizing sites holding the concurrent versions of `course-docs` and `student-projects`. The results confirm that the low values of locally untrackable redundancy have, as expected, a low impact on the effective transferred volume (data+metadata) and performance.

Space Requirements of Version Logs. Since redFS can only detect redundancy across the versions to send ($\mathcal{T}$) and the common version set ($\mathcal{C}$), the larger such an intersection is, the more redundancy redFS will be able to exploit. An assumption made so far is that the version logs at each synchronizing site could stretch arbitrarily. In practice, however, available log space is bounded. Hence, our ability to efficiently log versions determines how deep can the logged history at each replica be. The deeper the logged history, the higher the probability of larger $\mathcal{C}$ sets during synchronization with other sites.

To evaluate the impact of bounded log space on redFS's efficiency, we now consider multi-version variants of `course-docs` and `student-projects`.[5] For both workloads, we have obtained an exhaustive version set, comprising daily snapshots of the corresponding CVS repositories. Each version sequence spans across the whole duration of each workload: 168 days in `course-docs`, and 74 days in `student-projects`. Each version in the sequence exclusively includes the files whose contents have been modified relatively to the previous day's version.

For each workload, we assume that replica S has created every daily version of the workload, from day 0 to the last day, d, but may have already pruned the m oldest versions; hence, R stores every version from day $d - m$ to day d. In turn, replica R wishes to synchronize its stale replicas from S, as the last time S did that was in day d'. For simplicity, we assume R has enough space to log every version from day 0 to day d'.

When S and R synchronize, if $m > d'$, both replicas will share no common version (i.e., $\mathcal{C}$ is empty). Hence, in this case, S will detect no redundancy at all between the versions to send to R and the contents that R already stores.

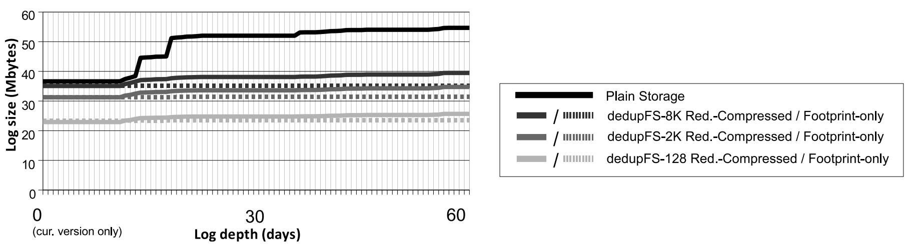

Fig. 3. Space requirements of version log for multi-version `student-projects` workload

Of course, S can only ensure sufficiently deep logs if their space requirements are below the available log space. Hence, we need to evaluate what are the space log requirements at S for increasingly deep logs. We have studied such requirements in both `course-docs` and `student-projects` workloads, considering the different schemes for log storage that Section 4 addresses: plain storage, redundancy compression and footprints, with different expected chunk sizes. Figure 3 presents such results for the latter workload.

A first observation is that, even with plain log storage, space cost grows only moderately with log depth. With `course-docs` (resp. `student-projects`), the space cost associated with a plain log is of additional 16.3% (resp. 16.3%) disk space than the most recent version, which requires 102 MB (resp. 37 MB), per logged month of update activity. Furthermore, more efficient log storage schemes are still able to substantially reduce such a space overhead. For both workloads, using redundancy compression, redFS can maintain the entire version history of 3 and 6 months, respectively, at the average cost of 4% space overhead per logged month of update activity. Finally, footprints offer almost negligible overhead, dropping to levels below 0.7% space overhead per logged month.

[5] Daily snapshots were not available for the other workloads.

7 Related Work

Significant work has addressed the exploitation of duplicate contents for improved storage and network dissemination of state-based updates. The main focus of our work is on the latter. We can divide most existing solutions to it into three main categories: delta-encoding, cooperating caches and compare-by-hash.

Systems such as SVN [11], Porcupine [12], BitKeeper [13], XDFS [14] and Git [15] use *delta-encoding* (or *diff-encoding*) to represent successive file versions. Starting from a base version of a file, they encode successive versions by a compact set of value-based deltas to the value of the preceding version. Systems that use delta encoding for distributed file transfer need to maintain some version tracking state at the communicating sites, similarly to our approach.

Our approach extends delta-encoding, improving the potential efficiency gains of the latter. Whereas delta-encoding is limited to locally untrackable redundancy between pairs of consecutive versions of the same file, we are able to exploit any cross-version and cross-object locally trackable redundancy.

A different technique [25] relies on a pair of cooperating caches maintained by a pair of sites. At any moment, both caches hold the same ordered set of n fixed-size blocks of data, which correspond to the last blocks propagated between the sites. Hence, before sending a new block, the sender checks whether its local cache already holds the block. If so, the site sends a token identifying the position of the block within the cache, instead of the block's contents.

Our approach shares the same principle, based on a conservative estimation of a common version/value set between the communicating sites. From such an estimate, both approaches exploit locally trackable cross-version and cross-object redundancy. Nevertheless, our approach scales gracefully to large numbers of sites and is able to cover a practically unlimited common version set; whereas cooperating caches need one cache per site and limit the common version set by cache size. Furthermore, cooperating caches cannot detect situations of transitive redundancy, whereas we can; i.e. when site A knows that it shares some chunk, c, with site B, and site B knows that it shares c with a third site, C, neither A nor C infer that both share c when they synchronize.

Finally, recent work [3,4,7,26,27] has followed the compare-by-hash approach for distributed data deduplication. They either divide values into contiguous (non-overlapping) variable-sized chunks, using content-based chunk division algorithms [4], or fixed-size chunks [3]. Compare-by-hash is able to detect both locally trackable and untrackable redundancy. Nevertheless, compare-by-hash has the important shortcomings discussed in Section 1.

Some work has proposed intricate variations of the technique for higher efficiency. The TAPER replicated system [8] optimizes bandwidth efficiency of compare-by-hash with a four-phase protocol, each of which works on a finer similarity granularity. Each phase works on the value portions that, after the preceding phase, remain labeled as non-redundant. A first phase detects larger-granularity redundant chunks (whole file and whole directories), using a hash tree; the Jumbo Store [10] distributed utility service employs a similar principle in a multi-phase protocol. A second phase runs the base compare-by-hash technique with content-based chunks. A third phase identifies pairs of very similar replicas (at each site) and employs rsync's technique upon each pair. Finally, a fourth phase employs delta-encoding to the remaining data.

8 Conclusions

We propose a novel technique for distributed locally trackable data deduplication. When compared with the prominent solutions for the distributed data deduplication problem, namely delta encoding and compare-by-hash, our solution borrows most advantages that distinguish each such alternative.

The results presented herein, obtained from real workloads, complement previous evidence [28] that contradicts a common conviction that, to some extent, is subjacent to most literature proposing compare-by-hash [4,7,8,10,9]. More precisely, our results show that, for the very workloads that the latter works consider and evaluate, there does exist a solution based exclusively on locally trackable deduplication that outperforms compare-by-hash. Perhaps more surprisingly, we show that even in scenarios with clear sources of locally untrackable redundancy, the impact of locally untrackable redundancy can be negligible. Consequently, our solution still outperforms compare-by-hash in such cases.

While it is easy to devise multiple-writer scenarios where locally untrackable redundancy prevails (e.g. two remote users that frequently copy identical data from a common external source, such a web site or an email server, to their local replicas of a object that both share), our work identifies important scenarios where it is advantageous to opt for locally trackable deduplication (which includes approaches such as delta-encoding, cooperative caching and ours) instead of compare-by-hash. More precisely, these include single-writer multiple-reader scenarios as long as sufficient disk space for logs is available, hence all redundancy is locally trackable. The same holds in multiple-writer scenarios where locally untrackable redundancy is negligible; our experience with workloads from real collaborative work scenarios suggests that this is often the case.

Acknowledgements

We would like to thank the anonymous reviewers, Lucy Cherkasova (our shepherd), Diogo Paulo and João Paiva for their helpful comments on previous versions of this paper.

References

1. Saito, Y., Shapiro, M.: Optimistic replication. ACM Computing Surveys 37(1), 42–81 (2005)
2. Dahlin, M., Chandra, B., Gao, L., Nayate, A.: End-to-end wan service availability. IEEE/ACM Transactions on Networking 11(2), 300–313 (2003)
3. Trigdell, A., Mackerras, P.: The rsync algorithm. Technical report, Australian National University (1998)
4. Muthitacharoen, A., Chen, B., Mazieres, D.: A low-bandwidth network file system. In: 8th ACM Symposium on Operating Systems Principles (SOSP), pp. 174–187 (2001)

5. Lelewer, D., Hirschberg, D.: Data compression. ACM Computing Surveys 19(3), 261–296 (1987)
6. Levy, E., Silberschatz, A.: Distributed file systems: Concepts and examples. ACM Computing Surveys 22(4), 321–374 (1990)
7. Cox, L., Noble, B.: Pastiche: Making backup cheap and easy. In: 5th Symposium on Operating Systems Design and Implementation, pp. 285–298. ACM, New York (2002)
8. Jain, N., Dahlin, M., Tewari, R.: Taper: Tiered approach for eliminating redundancy in replica sychronization. In: 4th USENIX FAST, p. 21 (2005)
9. Bobbarjung, D., Jagannathan, S., Dubnicki, C.: Improving duplicate elimination in storage systems. ACM Transactions on Storage 2(4), 424–448 (2006)
10. Eshghi, K., Lillibridge, M., Wilcock, L., Belrose, G., Hawkes, R.: Jumbo store: providing efficient incremental upload and versioning for a utility rendering service. In: 5th USENIX conference on File and Storage Technologies (FAST), p. 22 (2007)
11. Pilato, C., Fitzpatrick, B., Collins-Sussman, B.F.: Version Control with Subversion. O'Reilly, Sebastopol (2004)
12. Saito, Y., Bershad, B.N., Levy, H.M.: Manageability, availability, and performance in porcupine: a highly scalable, cluster-based mail service. ACM Trans. Comput. Syst. 18(3), 298 (2000)
13. Henson, V., Garzik, J.: Bitkeeper for kernel developers (2002), `http://infohost.nmt.edu/~val/ols/bk.ps.gz`
14. MacDonald, J.: File system support for delta compression. Masters thesis, University of California at Berkeley (2000)
15. Lynn, B.: Git magic (2009), `http://www-cs-students.stanford.edu/~blynn/gitmagic/`
16. Policroniades, C., Pratt, I.: Alternatives for detecting redundancy in storage systems data. In: USENIX Annual Technical Conference (2004)
17. Quinlan, S., Dorward, S.: Venti: A new approach to archival data storage. In: 1st USENIX Conference on File and Storage Technologies (FAST), p. 7 (2002)
18. Aronovich, L., Asher, R., Bachmat, E., Bitner, H., Hirsch, M., Klein, S.T.: The design of a similarity based deduplication system. In: ACM SYSTOR, pp. 1–14 (2009)
19. Petersen, K., Spreitzer, M., Terry, D., Theimer, M., Demers, A.: Flexible update propagation for weakly consistent replication. In: ACM SOSP, pp. 288–301 (1997)
20. Kistler, J.J., Satyanarayanan, M.: Disconnected operation in the coda file system. SIGOPS Oper. Syst. Rev. 25(5), 213–225 (1991)
21. Barreto, J.: Optimistic Replication in Weakly Connected Resource-Constrained Environments. PhD thesis, IST, Technical University Lisbon (2008)
22. Santry, D., Feeley, M., Hutchinson, N., Veitch, A., Carton, R., Ofir, J.: Deciding when to forget in the elephant file system. In: ACM SOSP, pp. 110–123 (1999)
23. Szeredi, M.: FUSE: Filesystem in Userspace (2008), `http://sourceforge.net/projects/avf`
24. MacDonald, J.: xdelta, `http://code.google.com/p/xdelta/`
25. Spring, N.T., Wetherall, D.: A protocol-independent technique for eliminating redundant network traffic. SIGCOMM Comput. Comm. Rev. 30(4), 87–95 (2000)
26. Tolia, N., Kozuch, M., Satyanarayanan, M., Karp, B., Perrig, A., Bressoud, T.: Opportunistic use of content addressable storage for distributed file systems. In: USENIX Annual Technical Conference, pp. 127–140 (2003)
27. Annapureddy, S., Freedman, M.J., Mazières, D.: Shark: scaling file servers via cooperative caching. In: USENIX Symp. Net. Sys. Design & Impl., pp. 129–142 (2005)
28. Henson, V.: An analysis of compare-by-hash. In: USENIX Workshop on Hot Topics in Operating Systems (2003)