Storage Tradeoffs in a Collaborative Backup Service for Mobile Devices[†]

Ludovic Courtès Marc-Olivier Killijian David Powell

first-name.last-name@laas.fr
LAAS-CNRS
7 avenue du Colonel Roche
31077 Toulouse cedex 4
France

Abstract

Mobile devices are increasingly relied on but are used in contexts that put them at risk of physical damage, loss or theft. We consider a fault-tolerance approach that exploits spontaneous interactions to implement a collaborative backup service. We define the constraints implied by the mobile environment, analyze how they translate into the storage layer of such a backup system and examine various design options. The paper concludes with a presentation of our prototype implementation of the storage layer, an evaluation of the impact of several compression methods, and directions for future work.

1. Introduction

Embedded computers are becoming widely available, in various portable devices such as PDAs, digital cameras, music players and laptops. Most of these devices are now able to communicate using wireless network technologies such as IEEE 802.11, Bluetooth, or Zigbee. Users use such devices to capture more and more data and are becoming increasingly dependent on them. Backing up the data stored on these devices is often done in an *ad hoc* fashion: each protocol and/or application has its own synchronization facilities that can be used when a sister device, usually a desktop computer, is reachable. However, newly created data may be held on the mobile device for a long time before it can be copied. This may be a serious issue since the contexts in which mobile devices are used increase the risks of them being lost, stolen or broken.

Our goal is to leverage the ubiquity of communicating mobile devices to implement a *collaborative* backup service. In such a system, devices participating in the service would be able to use other devices' storage to back up their own data. Of course, each device would have to contribute some of its own storage resources for others to be able to benefit from the service.

Internet-based peer-to-peer systems paved the way for such services. They showed that excess resources available at the peer hosts could be leveraged to support wide-scale resource sharing. Although the amount of resources available on a mobile device is significantly smaller than that of a desktop machine, we believe that this is not a barrier to the creation of mobile peer-to-peer services. They have also shown that wide-scale services could be created without relying on any infrastructure (other than the Internet itself), in a decentralized, self-administered way. From a fault-tolerance viewpoint, peer-to-peer systems provide a high diversity of nodes with independent failure modes [13].

In a mobile context, we believe there are additional reasons to use a collaborative service. For instance, access to a cell phone communication infrastructure (GPRS, UMTS, etc.) may be costly (especially for non-productive data transmission "just" for the sake of backup) while proximity communications are not (using 802.11, Bluetooth, etc.). Similarly, short-distance communication technologies are often more efficient than long-distance ones: they offer a higher throughput and often require less energy. In some scenarios, infrastructure-based networks are simply not available but neighboring devices might be accessible using single-hop communications, or by *ad hoc* routing.

Our target service raises a number of interesting issues, in particular relating to trust management, resource accounting and cooperation incentives. It raises novel issues due to, for instance, mostly-disconnected operation and the consequent difficulty of resorting to centralized

[†] This work was partially supported by the MoSAIC project (ACI S&I, French national program for Security and Informatics; see *http://www.laas.fr/mosaic/*) and the Hidenets project (EU-IST-FP6-26979).

or on-line solutions. A preliminary analysis of these issues may be found in [6,14]. In this paper, the focus is on the mechanisms employed at the storage layer of such a service. We investigate the various design options at this layer and discuss potential trade-offs.

In Section 2, we will detail the requirements of the cooperative backup service on the underlying storage layer. Section 3 presents several design options for this layer based on the current literature and the particular needs that arise from the kind of devices we target. In Section 4, using a prototype of this storage layer, we will evaluate some storage layer algorithms and discuss the necessary tradeoffs. Finally, we will conclude on our current work and sketch future research directions.

2. Collaborative Backup for Mobile Devices

This section gives an overview of the service envisaged and related work. Then we describe the requirements we have identified for the storage layer of the service.

2.1. Design Overview and Related Work

Our goal is to design and implement a collaborative backup system for communicating mobile devices. In this model, mobile devices can play the role of a *contributor*, i.e., a device that offers its storage resources to store data on behalf of other nodes, and a *data owner*, i.e., a mobile device asking a contributor to store some of its data on its behalf. Practically, nodes are expected to contribute as much as they benefit from the system; therefore, they should play both roles at the same time.

For the service to effectively leverage the availability of neighboring communicating devices, the service has to be functional even in the presence of *mutually suspicious device users*. We want users with no prior trust relationships to be able to use the service and to contribute to it harmlessly. This is in contrast with traditional habits where users usually back up their mobile devices' data only on machines they trust, such as their workstation.

This goal also contrasts with previous work on collaborative backup for a personal area network (PAN), such as FlashBack [19], where participating devices are all trustworthy since they belong to the same user. However, censorship-resistant peer-to-peer file sharing systems such as GNUnet [2] have a similar approach to security in the presence of adversaries.

Recently, a large amount of research has gone into the design and implementation of Internet-based peer-to-peer backup systems that do not assume prior trust relationships among participants [1,7,9]. There is, however, a significant difference between those Internet-based systems and what we envision: *connectivity*. Although these Internet-based collaborative backup systems are designed to tolerate disconnections, they do assume a high-level of connectivity. Disconnections are assumed to be mostly transient, whether they be non-malicious (a peer goes off-line for a few days or crashes) or malicious (a peer purposefully disconnects in order to try to benefit from the system without actually contributing to it).

In the context of mobile devices interacting spontaneously, connections are by definition short-lived, unpredictable, and very variable in bandwidth and reliability. Worse than that, a pair of peers may have a chance encounter and start exchanging data, and then never meet again.

To tackle this issue, we assume that each mobile device can at least *intermittently* access the Internet. The backup software running in those mobile devices is expected to take advantage of such an opportunity by re-establishing contacts with (proxies of) mobile devices encountered earlier. For instance, a contributor may wish to send data stored on behalf of another node to some sort of *repository* associated with the owner of the data. Contributors can thus asynchronously *push* data back to their owners. The repository itself can be implemented in various ways: an email mailbox, an FTP server, a fixed peer-to-peer storage system, etc. Likewise, data owners may sometimes need to query their repository as soon as they can access the Internet in order to *pull* back (i.e., restore) their data.

In the remainder of this paper, we will focus on the design of the storage layer of this service on both the data owner and contributor sides.

2.2. Requirements of the Storage Layer

We have identified the following requirements for the mechanisms employed at the storage layer.

Storage efficiency. Backing up data should be as efficient as possible. Data owners should neither ask contributors to store more data than necessary nor send excessive data over the wireless interface. Failing to do so will waste energy and result in inefficient utilization of the storage resources available in the node's vicinity. Inefficient storage may have a strong impact on energy consumption since (i) storage costs translate into transmission costs and (ii) energy consumption on mobile devices is dominated by wireless communication costs, which in turn increase as more data are transferred [28]. *Compression techniques* are thus a key aspect of the storage layer on the data owner side.

Small data blocks. Both the occurrence of encounters of a peer within radio range and the lifetime of the resulting connections are unpredictable. Consequently, the backup application running on a data owner's device

must be able to conveniently split the data to be backed up into small pieces to ensure that it can actually be transferred to contributors. Ideally, data blocks should be able to fit within the underlying network layer's maximum transmission unit or MTU (2304 octets for IEEE 802.11); larger payloads get fragmented into several packets, which introduces overhead at the MAC layer, and possibly at the transport layer too.

Backup atomicity. Unpredictability and the potentially short lifetime of connections, compounded with the possible use of differential compression to save storage resources, mean that it is unlikely to be practical to store a set of files, or even one complete file, on a single peer. Indeed, it may even be undesirable to do so in order to protect data confidentiality [8]. Furthermore, it may be the case that files are modified before their previous version has been completely backed up.

The dissemination of data chunks as well as the coexistence of several versions of a file must not affect backup consistency as perceived by the end-user: a file should be either retrievable *and* correct, or unavailable. Likewise, the distributed store that consists of various contributors shall remain in a "legal" state after new data are backed up onto it. This corresponds to the *atomicity* and *consistency* properties of the ACID properties commonly referred to in transactional database management systems.

Error detection. Accidental modifications of the data are assumed to be handled by the various lower-level software and hardware components involved, such as the communication protocol stack, the storage devices themselves, the operating system's file system implementation, etc. However, given that data owners are to hand their data to untrusted peers, the storage layer must provide mechanisms to ensure that *malicious* modifications to their data are detected with a high probability.

Encryption. Due to the lack of trust in contributors, data owners may wish to encrypt their data to ensure privacy. While there exist scenarios where there is sufficient trust among the participants such that encryption is not compulsory (e.g., several people in the same working group), encryption is a requirement in the general case.

Backup redundancy. Redundancy is the *raison d'être* of any data backup system, but when the system is based on cooperation, the backups themselves must be made redundant. First, the cooperative backup software must account for the fact that contributors may crash accidently. Second, contributor availability is unpredictable in a mobile environment without continuous Internet access. Third, contributors are not fully trusted and may behave maliciously. Indeed, the literature on Internet-based peer-to-peer backup systems describes a range of attacks against data availability, ranging from data retention (i.e., a contributor purposefully refuses to allow a data owner to retrieve its data) to selfishness (i.e., a participant refuses to spend energy and storage resources storing data on behalf of other nodes) [7,9]. All these uncertainties make redundancy even more critical in a cooperative backup service for mobile devices.

3. Design Options for the Storage Layer

In this section, we present design options able to satisfy each of the requirements identified for above.

3.1. Storage Efficiency

In wired distributed cooperative services, storage efficiency is often addressed by ensuring that a given content is only stored once. This property is known as *single-instance storage* [4]. It can be thought of as a form of compression among several data units. In a file system, where the "data unit" is the file, this means that a given content stored under different file names will be stored only once. On Unix-like systems, revision control and backup tools implement this property by using hard links [20,25]. It may also be provided at a sub-file granularity, instead of at a whole file level, allowing reduction of unnecessary duplication with a finer-grain.

Archival systems [23,35], peer-to-peer file sharing systems [2], peer-to-peer backup systems [7], network file systems [22], and remote synchronization tools [31] have been demonstrated to benefit from single-instance storage, either by improving storage efficiency or reducing bandwidth.

Compression based on resemblance detection, i.e., *differential compression*, or *delta encoding*, is unsuitable for our environment since (i) it requires access to all the files already stored, (ii) it is CPU- and memory-intensive, and (iii) the resulting *delta chains* weaken data availability [15,35].

Traditional lossless compression (i.e., *zip* variants), allows the elimination of duplication *within* single files. As such, it naturally complements inter-file and inter-version compression techniques [35]. Section 4 contains a discussion of the combination of both techniques in the framework of our proposed backup service. Lossless compressors usually yield better compression when operating on large input streams [15] so compressing concatenated files rather than individual files improves storage efficiency [35]. However, we did not consider this approach suitable for mobile device backup since it may be more efficient to backup only those files (or part of files) that have changed.

There exist a number of application-specific compression algorithms, such as the *lossless* algorithms used

by the FLAC audio codec, the PNG image format, and the XMill XML compressor [17]. There is also a plethora of *lossy* compression algorithms for audio samples, images, videos, etc. While using such application-specific algorithms might be beneficial in some cases, we have focused instead on generic lossless compression.

3.2. Small Data Blocks

We now consider the options available to: (1) chop input streams into small blocks, and (2) create appropriate meta-data describing how those data blocks should be reassembled to produce the original stream.

3.2.1. Chopping Algorithms

As stated in Section 2.2, the size of blocks that are to be sent to contributors of the backup service has to be bounded, and preferably small, to match the nature of peer interactions in a mobile environment. There are several ways to cut input streams into blocks. Which algorithm is chosen has an impact on the improvement yielded by single-instance storage applied at the block level.

Splitting input streams into fixed-size blocks is a natural solution. When used in conjunction with a single-instance storage mechanism, it has been shown to improve the compression across files or across file versions [23]. Manber proposed an alternative content-based stream chopping algorithm [21] that yields better duplication detection across files, a technique sometimes referred to as *content-defined blocks* [15]. The algorithm determines block boundaries by computing Rabin fingerprints on a sliding window of the input streams. Thus, it only allows the specification of an *average* block size (assuming random input). Various applications such as archival systems [35], network file systems [22] and backup systems [7] benefit from this algorithm. Section 4 provides a comparison of both algorithms.

3.2.2. Stream Meta-Data

Placement of stream meta-data. Stream meta-data is information that describes which blocks comprise the stream and how they should be reassembled to produce the original stream. Such meta-data can either be embedded along with each data block or stored separately. The main evaluation criteria of a meta-data structure are read efficiency (e.g., algorithmic complexity of stream retrieval, number of accesses needed) and size (e.g., how the amount of meta-data grows compared to data).

We suggest a more flexible approach whereby stream meta-data (i.e., which blocks comprise a stream) is separated both from file meta-data (i.e., file name, permissions, etc.) and the file content. This has several advantages. First, it allows a data block to be referenced multiple times and hence allows for single-instance storage at the block level. Second, it promotes *separation of concerns*. For instance, file-level meta-data (e.g., file path, modification time, permissions) may change without having to modify the underlying data blocks, which is important in scenarios where propagating such updates would be next to impossible. Separating meta-data and data also leaves the possibility of applying the same "filters" (e.g., compression, encryption), or to use similar redundancy techniques for both data and meta-data blocks. This will be illustrated in Section 4. This approach is different from the one used in Hydra [34] but not unlike that of OpenCM [27].

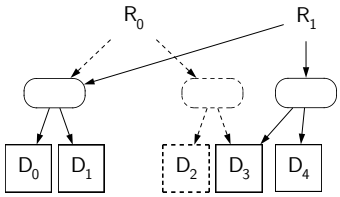

Figure 1. A tree structure for stream meta-data. Leaves represent data blocks while higher blocks are meta-data blocks.

Indexing individual blocks. The separation of data and meta-data means that there must be a way for meta-data blocks to refer to data blocks: data blocks must be indexed or *named*[1]. The block naming scheme must fulfill several requirements. First, it must not be based on non-backed-up user state which would be lost during a crash. Most importantly, the block naming scheme must guarantee that *name clashes* among the blocks of a data owner cannot occur. In particular, block IDs must remain valid in time so that a given block ID is not wrongfully re-used when a device restarts the backup software after a crash. Given that data blocks will be disseminated among several peers and will ultimately migrate to their owner's repository, blocks IDs should remain valid in space, that is, they should be independent of contributor names. This property also allows for *precomputation* of block IDs and meta-data blocks: stream chopping and indexing do not need to be done upon a contributor encounter, but can be performed *a priori*, once for all. This saves CPU time and energy, and allows data owners to immediately take advantage of a backup opportunity. A practical naming scheme widely used in the literature will be discussed in Section 3.4.

Indexing sequences of blocks. Byte streams (file contents) can be thought of as sequences of blocks.

[1] In the sequel we use the terms "block ID", "name", and "key" interchangeably.

Meta-data describing the list of blocks comprising a byte stream need to be produced and stored. In their simplest form, such meta-data are a vector of block IDs, or in other words, a byte stream. This means that this byte stream can in turn be indexed, recursively, until a meta-data byte stream is produced that fits the block size constraints. This approach yields the meta-data structure shown in Figure 1 which is comparable to that used by Venti and GNUnet [2,23].

Contributor interface. With such a design, contributors do not need to know about the actual implementation of block and stream indexing used by their clients, nor do they need to be aware of the data/meta-data distinction. All they need to do is to provide primitives of a keyed block storage:

- `put (key, data)` inserts the data block `data` and associates it with `key`, a block ID chosen by the data owner according to some naming scheme;
- `get (key)` returns the data associated with `key`.

This simple interface suffices to implement, on the data owner side, byte stream indexing and retrieval. Also, it is suitable for an environment in which service providers and users are mutually suspicious because it places as little burden as possible on the contributor side. The same approach was adopted by Venti [23] and by many peer-to-peer systems [2,7].

3.3. Backup Atomicity

Distributed and mobile file systems such as Coda [16] which support concurrent read-write access to the data and do not have built-in support for revision control, differ significantly from backup systems. Namely, they are concerned about update propagation and reconciliation in the presence of concurrent updates. Not surprisingly, a read-write approach does not adapt well to the loosely connected scenarios we are targeting: data owners are not guaranteed to meet *every* contributor storing data on their behalf in a timely fashion, which makes update propagation almost impossible. Additionally, it does not offer the desired atomicity requirement discussed in Section 2.2.

The *write once* or *append only* semantics adopted by archival [11,23], backup [7,25] and versioning systems [20,26,27] solve these problems. Data is always appended to the storage system, and never modified in place. This is achieved by assigning each piece of data a unique identifier. Therefore, insertion of content (i.e., data blocks) into the storage mechanism (be it a peer machine, a local file system or data repository) is atomic. Because data is only added, never modified, consistency is also guaranteed: insertion of a block cannot result in an inconsistent state of the storage mechanism.

A potential concern with this approach is its cost in terms of storage resources. It has been argued, however, that the cost of storing subsequent revisions of whole sets of files can be very low provided inter-version compression techniques like those described earlier are used [10,23,26]. In our case, once a contributor has finally transferred data to their owner's repository, it may reclaim the corresponding storage resources, which further limits the cost of this approach.

From an end-user viewpoint, being able to restore an old copy of a file is more valuable than being unable to restore the file at all. All these reasons make the write-only approach suitable to the storage layer of our cooperative backup service.

3.4. Error Detection

Error-detecting codes can be computed either at the level of whole input streams or at the level of data blocks. They must then be part of, respectively, the stream meta-data, or the block meta-data. We argue the case for cryptographic hash functions as a means of providing the required error detection and as a block-level indexing scheme.

Cryptographic hash functions. The error-detecting code we use must be able to detect *malicious* modifications. This makes error-detecting codes designed to tolerate random, accidental faults inappropriate. We must instead use *collision-resistant* and *preimage-resistant* hash functions, which are explicitly designed to detect tampering [5].

Along with integrity, *authenticity* of the data must also be guaranteed, otherwise a malicious contributor could deceive a data owner by producing fake data blocks along with valid cryptographic hashes. Thus, digital signatures should be used to guarantee the authenticity of the data blocks. Fortunately, not all blocks need to be signed: signing a root meta-data block (as shown in Figure 1) is sufficient. This is similar to the approach taken by OpenCM [27]. Note, however, that while producing random data blocks and their hashes is easy, producing the corresponding meta-data blocks is next to impossible without knowing what particular meta-data schema is used by the data owner.

Content-based indexing. Collision-resistant hash functions have been assumed to meet the requirements of a data block naming scheme as defined in Section 3.2.2, and to be a tool allowing for efficient implementations of single-instance storage [7,22,23,29,31,35]. In practice, these implementations assume that whenever two data blocks yield the same cryptographic hash value, their contents *are* identical. Given this assumption, implementation of a single-instance store is straightforward: a block only needs to be stored if its hash val-

ue was not found in the locally maintained block hash table.

In [12], Henson argues that accidental collisions, although extremely rare, do have a slight negative impact on software reliability and yield silent errors. Given an n-bit hash output produced by one of the functions listed above, the expected workload to generate a collision out of two *random* inputs is of the order of $2^{n/2}$ [5]. More precisely, if we are to store, say, 8 GiB of data in the form of 1 KiB blocks, we end up with 2^{43} blocks, whereas SHA-1, for instance, would require 2^{80} blocks to be generated on average before an accidental collision occurs. We consider this to be reasonable in our application since it does not impede the tolerance of faults in any significant way. Also, Henson's fear of *malicious* collisions does not hold given the preimage-resistance property provided by the commonly-used hash functions[2].

Content-addressable storage (CAS) thus seems a viable option for our software layer as it fulfills both the error-detection and data block naming requirements. In [29], the authors assume a block ID space shared across all CAS users and providers. In our scenario, CAS providers (contributors) do not trust their clients (data owners) so they need either to enforce the block naming scheme (i.e., make sure that the ID of each block is indeed the hash value of its content), or to maintain a per-user name space.

3.5. Encryption

Data encryption may be performed either at the level of individual blocks, or at the level of input streams. Encrypting the input stream *before* it is split into smaller blocks breaks the single-instance storage property at the level of individual blocks. This is because encryption aims to ensure that the encrypted output of two similar input streams will not be correlated.

Leaving input streams unencrypted and encrypting individual blocks yielded by the chopping algorithm does not have this disadvantage. More precisely, it preserves single-instance storage at the level of blocks at least *locally*, i.e., on the client side. If asymmetric ciphering algorithms are used, the single-instance storage property is no longer ensured *across* peers, since each peer encrypts data with its own private key. However, we do not consider this a major drawback for the majority of scenarios considered where little or no data are common to several participants. Moreover, solutions to this problem exist, notably *convergent encryption* [7].

[2]The recent attacks found on SHA-1 by Wang et al. [33] do not affect the preimage-resistance of this function.

3.6. Backup Redundancy

Replication strategies. Redundancy management in the context of our collaborative backup service for mobile devices introduces a number of new challenges. Peer-to-peer file sharing systems are not a good source of inspiration in this respect given that they rely on redundancy primarily as a means of reducing access time to popular content [24].

For the purposes of fault-tolerance, statically-defined redundancy strategies have been used in Internet-based scenarios where the set of servers responsible for holding replicas is known *a priori*, and where servers are usually assumed to be reachable "most of the time" [8,34]. Internet-based peer-to-peer backup systems [7,9] have relaxed these assumptions. However, although they take into account the fact that contributors may become unreachable, strong connectivity assumptions are still made: the inability to reach a contributor is assumed to be the exception, rather than the rule. As a consequence, unavailability of a contributor is quickly interpreted as a symptom of malicious behavior [7,9].

The connectivity assumption does not hold in our case. Additionally, unlike with Internet-based systems, the very encounter of a contributor is unpredictable. This has a strong impact on the possible replication strategies, and on the techniques used to implement redundancy.

Erasure codes have been used as a means to tolerate failures of storage sites while being more storage-efficient than simple replication [34]. Usually, (n,k) erasure codes are defined as follows [18,34]:

- an (n,k) code maps a k-symbol block to an n-symbol codeword;
- $k + \varepsilon$ symbols suffice to recover the exact original data; the code is *optimal* when $\varepsilon = 0$;
- optimal (n,k) schemes tolerate the loss of $(n - k)$ symbols and have an effective storage use of k/n.

Such an approach seems very attractive to improve storage efficiency while still maximizing data availability.

However, as argued in [3,18,32], an (n,k) scheme with $k > 1$ can hinder data availability because it requires k peers to be available for data to be retrieved, instead of just 1 with mirroring (i.e., an $(n,1)$ scheme). Also, given the unpredictability of contributor encounters, a scheme with $k > 1$ is risky since a data owner may fail to store k symbols on different contributors. On the other hand, from a confidentiality viewpoint, increasing dissemination and purposefully placing less than k symbols on any given untrusted contributor may be a good strategy [8]. Intermediate solutions can also be imagined, e.g., mirroring blocks that have never been replicated and choosing

$k > 1$ for blocks already mirrored at least once. This use of different *levels of dispersal* was also mentioned by the authors of InterMemory [11] as a way to accommodate contradictory requirements. Finally, a dynamically adaptive behavior of erasure coding may be considered as [3] suggests.

Replica scheduling and dissemination. As stated in Section 2.2, it is plausible that a file will be only partly backed up when a newer version of this file enters the backup creation pipeline. One could argue that the replica scheduler should finish distributing the data blocks from the old version before distributing those of the new version. This policy would guarantee, at least, availability of the old version of the file. On the other hand, in certain scenarios, users might want to favor freshness over availability, i.e., they might request that newer blocks are scheduled first for replication.

This clearly illustrates that a wide range of *replica scheduling and dissemination policies and corresponding algorithms* can be defended depending on the scenario considered. At the core of a given replica scheduling and dissemination algorithm is a *dispersal function* that decides on a level of dispersal for any given data block. The algorithm must evolve *dynamically* to account for several changing factors. In FlashBack [19], the authors identify a number of important factors that they use to define a *device utility function*. Those factors include *locality* (i.e., the likelihood of encountering a given device again later) as well as *power and storage resources* of the device.

In addition to those factors, our backup software needs to account for the current level of trust in the contributor at hand. If a data owner fully trusts a contributor, e.g., because it has proven to be well-behaved over a given period of time, the data owner may choose to store complete replicas (i.e., mirrors) on this contributor.

4. Preliminary Evaluation

This section presents our prototype implementation of the storage layer of the envisaged backup system, as well as a preliminary evaluation of key aspects.

4.1. Implementation Overview

We have implemented a prototype of the storage layer discussed above, a basic building block of the cooperative backup framework we are designing. This layer is performance-critical and we implemented it in C. The resulting library, `libchop`, consists of 7 K physical source lines of code. It was designed to be flexible enough so that different techniques could be combined and evaluated, by providing a few well-defined interfaces as shown in Figure 2. The library itself is not concerned

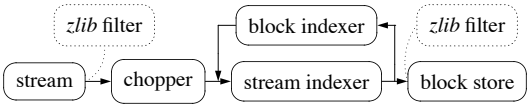

Figure 2. Data flow in the `libchop` backup creation pipeline.

with the backup of file system-related meta-data such as file paths, permissions, etc. Implementing this is left to higher-level layers akin to OpenCM's schemas [27].

Implementations of the `chopper` interface chop input streams into small fixed-size blocks, or according to Manber's algorithm [21]. Block indexers name blocks and store them in a keyed block store (e.g., an on-disk database). The `stream_indexer` interface provides a method that iterates over the blocks yielded by the given chopper, indexes them, produces corresponding meta-data blocks, and stores them in a block store. In the proposed cooperative backup service, chopping and indexing are to be performed on the data owner side, while the block store itself will be realized by contributors. Finally, `libchop` also provides *filters*, such as *zlib* compression and decompression filters, which may be conveniently reused in different places, for instance between a file-based input stream and a chopper, or between a stream indexer and a block store.

In the following experiments, the only stream indexer used is a "tree indexer" as shown in Figure 1. We used an on-disk block store that uses TDB as the underlying database [30]. For each file set, we started with a new, empty database.

4.2. Evaluation of Compression Techniques

Our implementation has allowed us to evaluate more precisely some of the tradeoffs outlined in Section 3. After describing the methodology and workloads that were used, we will comment the results obtained.

4.2.1. Methodology and Workloads

Methodology. The purpose of our evaluation is to compare the various compression techniques described earlier in order to better understand the tradeoffs that must be made. We measured the storage efficiency and computational cost of each method, both of which are critical criteria for resource-constrained devices. The measures were performed on a 500 MHz G4 Macintosh running GNU/Linux (running them on, say, an ARM-based mobile device would have resulted in lower throughputs; however, since we are interested in *comparing* throughputs, this would not make any significant difference).

Name	Size	Files	Avg. Size
Lout (versions 3.20 to 3.29)	76 MiB	5853	13 KiB
Ogg Vorbis files	69 MiB	17	4 MiB
mbox-formatted mailbox	7 MiB	1	7 MiB

Figure 3. File sets.

We chose several workloads and compared the results obtained using different configurations. These file sets, shown in Figure 3, qualify as *semi-synthetic* workloads because they are actual workloads, although they were not taken from a real mobile device. The motivation for this choice was to purposefully target specific file *classes*. The idea is that the results should remain valid for any file of these classes.

File sets. In Figure 3, the first file set contains 10 successive versions of the source code of the Lout document formatting system, i.e., low-density, textual input (C and Lout code), spread across a number of small files. Of course, this type of data is not typical of mobile devices like PDAs and cell phones. Nevertheless, the results obtained with this workload should be similar to those obtained with widely-used textual data format such as XML. The second file set shown in Figure 3 consists of 17 Ogg Vorbis files, a high-density binary format (Ogg Vorbis is a lossy audio compression format), typical of the kind of data that may be found on devices equipped with sampling peripherals. The third file set consists of a single, large file: a mailbox in the Unix mbox format which is an append-only textual format. Such data are likely to be found in a similar form on communicating devices.

Configurations. Figure 4 shows the storage configurations we have used in our experiments. For each configuration, it indicates whether single-instance storage was provided, which chopping algorithm was used and what the expected block size was, as well as whether the input stream or output blocks were compressed using a lossless stream compression algorithm (*zlib* in our case). Our intent is not to evaluate the outcome of each algorithm independently, but rather that of whole configurations. Thus, instead of experimenting with every possible combination, we chose to retain only those that (i) made sense from an algorithmic viewpoint and (ii) were helpful in understanding the tradeoffs at hand.

Configurations A_1 and A_2 serve as baselines for the overall compression ratio and computational cost. Comparing them is also helpful in determining the computational cost due to single-instance storage alone. Subsequent configurations all chop input streams into small blocks whose size fits our requirements (1 KiB, which should yield packets slightly smaller than IEEE 802.11's MTU); they all implement single-instance storage of the blocks produced.

Config.	Single Instance?	Chopping Algo.	Expected Block Size	Input Zipped?	Blocks Zipped?
A_1	no	—	—	yes	—
A_2	yes	—	—	yes	—
B_1	yes	Manber's	1024 B	no	no
B_2	yes	Manber's	1024 B	no	yes
B_3	yes	fixed-size	1024 B	no	yes
C	yes	fixed-size	1024 B	yes	no

Figure 4. Description of the configurations experimented.

Common octet sequences are unlikely to be found *within* a *zlib*-compressed stream, by definition. Hence, zipping the input precludes advantages to be gained by block-level single-instance storage afterwards. Thus, we did not include a configuration where a zipped input stream would then be passed to a chopper implementing Manber's algorithm.

The *B* configurations favor sub-file single-instance storage by not compressing the input before chopping it. B_2 improves over B_1 by adding the benefits of *zlib* compression at the block-level. Conversely, configuration *C* favors traditional lossless compression over sub-file single-instance storage since it applies lossless compression to the input stream.

Our implementation of Manber's algorithm uses a sliding window of 48 B which was reported to provide good results [22]. All configurations but A_1 use single-instance storage, realized using the `libchop` "hash" block indexer that uses SHA-1 hashes as unique block identifiers. For A_1, a block indexer that systematically provides unique IDs (per RFC 4122) was chosen.

The chosen configurations and file sets are quite similar to those described in [15,35], except that, as as explained in Section 3.1, we do not evaluate the storage efficiency of the delta encoding technique proposed therein.

4.2.2. Results

Figure 5 shows the compression ratios obtained for each configuration and each file set. The bars show the ratio of the size of the resulting blocks, *including* meta-data (sequences of SHA-1 hashes), to the size of the input data, for each configuration and each data set. The lines represent the corresponding throughputs.

Impact of the data type. Not suprisingly, the set of Vorbis files defeats all the compression techniques. Most configurations incur a slight storage overhead due to the amount of meta-data generated.

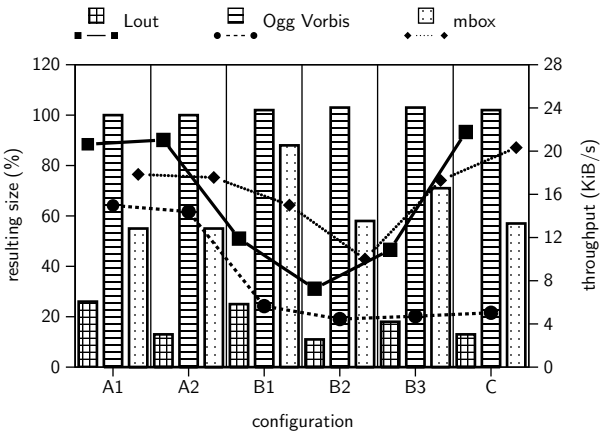

Figure 5. Storage efficiency and computational cost of several configurations.

Impact of single-instance storage. Comparing the results obtained for A_1 and A_2 shows benefits only in the case of the successive source code distributions, where it halves the amount of data stored (13 % vs. 26 %). This is due to the fact that successive versions of the software have a lot of files in common. Furthermore, it shows that single-instance storage implemented using cryptographic hashes does not degrade throughput, which is the reason why we chose to use it in all configurations.

As expected, single-instance storage applied at the block-level is mainly beneficial for the Lout file set where it achieves noticeable inter-version compression, comparable with that produced with *zlib* in A_1. The best compression ratio overall is obtained with B_2 where individual blocks are *zlib*-compressed. However, the compression ratios obtained with B_2 are comparable to those obtained with C, and only slightly better in the Lout case (11 % vs. 13 %). Thus, we conclude that there is little storage efficiency improvement to be gained from the combination of single-instance storage and Manber's chopping algorithm compared to traditional lossless compression, especially when applied to the input stream.

The results in [35] are slightly more optimistic regarding the storage efficiency of a configuration similar to B_2, which may be due to the use a smaller block (512 B) and a larger file set.

Computational cost. Comparing the computational costs of the B configurations with that of C provides an important indication as to which kind of configuration suits our needs best. Indeed, input zipping and fixed-size chopping in C yield a processing throughput three times higher than that of B_2 (except for the set of Vorbis files). Thus, C is the configuration that offers the best tradeoff between computational cost and storage efficiency for low-entropy data.

Additional conclusions can be drawn with respect to throughput. First, the cost of *zlib*-based compression appears to be reasonable, particularly when performed on the input stream rather than on individual blocks, as evidenced, e.g., by B_3 and C. Second, the input data type has a noticeable impact on the computational cost. In particular, applying lossless compression is more costly for the Vorbis files than for low-entropy data. Therefore, it would be worthwhile to disable *zlib* compression for compressed data types.

5. Conclusion and Future Work

In this paper, we have considered the viability of collaboration between peer mobile devices to implement a cooperative backup service. We have identified six essential requirements for the storage layer of such a service, namely: (i) storage efficiency; (ii) small data blocks; (iii) backup atomicity; (iv) error detection; (v) encryption; (vi) backup redundancy. The various design options for meeting these requirements have been reviewed and a preliminary evaluation carried out using a prototype implementation of the storage layer.

Our evaluation has allowed us to assess different storage techniques, both in terms of storage efficiency and computational cost. We conclude that the most suitable combination for our purposes combines the use of lossless input compression with fixed-size chopping and single-instance storage. Other techniques were rejected for providing little storage efficiency improvement compared to their CPU cost.

Future work on the optimization of the storage layer concerns several aspects. First, the energy costs of the various design options need to be assessed, especially those related to the wireless transmission of backup data between nodes. Second, the performance and dependability impacts of various replica scheduling and dissemination strategies need to be evaluated as a function, for example, of the expected frequencies of data updates, cooperative backup opportunities and infrastructure connections. Third, it seems likely that no single configuration of the backup service will be appropriate for all situations, so dynamic adaptation of the service to suit different contexts needs to be investigated.

Finally, the issues relating to trust management, resource accounting and cooperation incentives need to be addressed, especially insomuch as the envisaged mode of mostly-disconnected operation imposes additional constraints. Current research in this direction, in collaboration with our partners in the MoSAIC project, is directed at evaluating mechanisms such as microeconomic and reputation-based incentives.

References

[1] C. Batten, K. Barr, A. Saraf, S. Treptin. pStore: A secure peer-to-peer backup system. MIT-LCS-TM-632, MIT Laboratory for Computer Science, December 2001.

[2] K. Bennett, C. Grothoff, T. Horozov, I. Patrascu. Efficient Sharing of Encrypted Data. *Proc. of the 7th Australasian Conf. on Information Security and Privacy (ACISP 2002)*, (2384)pp. 107–120, 2002.

[3] R. Bhagwan, K. Tati, Y-C. Cheng, S. Savage, G. M. Voelker. Total Recall: System Support for Automated Availability Management. *Proc. of the ACM/USENIX NSDI*, 2004.

[4] W. J. Bolosky, J. R. Douceur, D. Ely, M. Theimer. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. *Proc. of the Int. Conf. on Measurement and Modeling of Computer Systems*, pp. 34–43, 2000.

[5] NESSIE Consortium. NESSIE Security Report. NES/DOC/ENS/WP5/D20/2, February 2003.

[6] L. Courtès, M-O. Killijian, D. Powell, M. Roy. Sauvegarde coopérative entre pairs pour dispositifs mobiles. *Actes des deuxièmes journées francophones Mobilité et Ubiquité (UbiMob)*, pp. 97–104, 2005.

[7] L. P. Cox, B. D. Noble. Pastiche: Making Backup Cheap and Easy. *5th USENIX OSDI*, pp. 285–298, 2002.

[8] Y. Deswarte, L. Blain, J-C. Fabre. Intrusion Tolerance in Distributed Computing Systems. *Proc. of the IEEE Symp. on Research in Security and Privacy*, pp. 110–121, 1991.

[9] S. Elnikety, M. Lillibridge, M. Burrows. Peer-to-peer Cooperative Backup System. *The USENIX FAST*, 2002.

[10] T. J. Gibson, E. L. Miller. Long-Term File Activity Patterns in a UNIX Workstation Environment. *Proc. of the 15th IEEE Symp. on MSS*, pp. 355–372, 1998.

[11] A. V. Goldberg, P. N. Yianilos. Towards an Archival Intermemory. *Proc. IEEE Int. Forum on Research and Technology Advances in Digital Libraries (ADL'98)*, pp. 147–156, 1998.

[12] V. Henson. An Analysis of Compare-by-hash. *Proc. of HotOS IX: The 9th HotOS*, pp. 13–18, 2003.

[13] F. Junqueira, R. Bhagwan, K. Marzullo, S. Savage, G. M. Voelker. The Phoenix Recovery System: Rebuilding from the Ashes of an Internet Catastrophe. *9th HotOS*, 2003.

[14] M-O. Killijian, D. Powell, M. Banâtre, P. Couderc, Y. Roudier. Collaborative Backup for Dependable Mobile Applications. *Proc. of 2nd Int. Workshop on Middleware for Pervasive and Ad-Hoc Computing (Middleware 2004)*, pp. 146–149, 2004.

[15] P. Kulkarni, F. Douglis, J. LaVoie, J. M. Tracey. Redundancy Elimination Within Large Collections of Files. *Proc. of the USENIX Annual Technical Conf.*, 2004.

[16] Y-W. Lee, K-S. Leung, M. Satyanarayanan. Operation-based Update Propagation in a Mobile File System. *Proc. of the USENIX Annual Technical Conf.*, pp. 43–56, 1999.

[17] H. Liefke, D. Suciu. XMill: an Efficient Compressor for XML Data. *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pp. 153–164, 2000.

[18] W. K. Lin, D. M. Chiu, Y. B. Lee. Erasure Code Replication Revisited. *Proc. of the 4th P2P*, pp. 90–97, 2004.

[19] B. T. Loo, A. LaMarca, G. Borriello. Peer-To-Peer Backup for Personal Area Networks. IRS-TR-02-015, UC Berkeley; Intel Seattle Research (USA), May 2003.

[20] T. Lord. The GNU Arch Distributed Revision Control System. 2005, *http://www.gnu.org/software/gnu-arch/*.

[21] U. Manber. Finding Similar Files in a Large File System. *Proc. of the USENIX Winter 1994 Conf.*, pp. 1–10, 1994.

[22] A. Muthitacharoen, B. Chen, D. Mazières. A Low-Bandwidth Network File System. *Proc. of the 18th ACM SOSP*, pp. 174–187, 2001.

[23] S. Quinlan, S. Dorward. Venti: A New Approach to Archival Storage. *Proc. of the 1st USENIX FAST*, pp. 89–101, 2002.

[24] K. Ranganathan, A. Iamnitchi, I. Foster. Improving Data Availability Through Dynamic Model-Driven Replication in Large Peer-to-Peer Communities. *Proc. of the Workshop on Global and Peer-to-Peer Computing on Large Scale Distributed Systems*, pp. 376–381, 2002.

[25] M. Rubel. Rsnapshot: A Remote Filesystem Snapshot Utility Based on Rsync. 2005, *http://rsnapshot.org/*.

[26] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, J. Ofir. Deciding when to forget in the Elephant file system. *Proc. of the 17th ACM SOSP*, pp. 110–123, 1999.

[27] J. S. Shapiro, J. Vanderburgh. CPCMS: A Configuration Management System Based on Cryptographic Names. *Proc. of the USENIX Annual Technical Conf., FREENIX Track*, pp. 207–220, 2002.

[28] M. Stemm, P. Gauthier, D. Harada, R. H. Katz. Reducing Power Consumption of Network Interfaces in Hand-Held Devices. *IEEE Transactions on Communications*, E80-B(8), August 1997, pp. 1125–1131.

[29] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. Bressoud, A. Perrig. Opportunistic Use of Content Addressable Storage for Distributed File Systems. *Proc. of the USENIX Annual Technical Conf.*, pp. 127–140, 2003.

[30] A. Tridgell, P. Russel, J. Allison. The Trivial Database. 1999, *http://samba.org/*.

[31] A. Tridgell, P. Mackerras. The Rsync Algorithm. TR-CS-96-05, Department of Computer Science, Australian National University Canberra, Australia, 1996.

[32] A. Vernois, G. Utard. Data Durability in Peer to Peer Storage Systems. *Proc. of the 4th Workshop on Global and Peer to Peer Computing*, pp. 90–97, 2004.

[33] X. Wang, Y. Yin, H. Yu. Finding Collisions in the Full SHA-1. *Proc. of the CRYPTO Conf.*, pp. 17–36, 2005.

[34] L. Xu. Hydra: A Platform for Survivable and Secure Data Storage Systems. *Proc. of the ACM Workshop on Storage Security and Survivability*, pp. 108–114, 2005.

[35] L. L. You, K. T. Pollack, and D. D. E. Long. Deep Store: An Archival Storage System Architecture. *Proc. of the 21st ICDE*, pp. 804–815, 2005.