SCFS: A Shared Cloud-backed File System

Alysson Bessani, Ricardo Mendes, Tiago Oliveira, and Nuno Neves, ***Faculdade de Ciências and LaSIGE;*** **Miguel Correia,** ***INESC-ID and Instituto Superior Técnico, University of Lisbon;*** **Marcelo Pasin,** ***Université de Neuchâtel;*** **Paulo Verissimo,** ***Faculdade de Ciências and LaSIGE***

https://www.usenix.org/conference/atc14/technical-sessions/presentation/bessani

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Open access to the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

SCFS: A Shared Cloud-backed File System

Alysson Bessani,[1] *Ricardo Mendes*,[1] *Tiago Oliveira*,[1] *Nuno Neves*,[1]
Miguel Correia,[2] *Marcelo Pasin*,[3] *Paulo Verissimo*[1]

[1]Faculdade de Ciências/LaSIGE, [2]Instituto Superior Técnico/INESC-ID, Universidade de Lisboa, Portugal

[3]University of Neuchatel, Switzerland

Abstract

Despite of their rising popularity, current cloud storage services and cloud-backed storage systems still have some limitations related to reliability, durability assurances and inefficient file sharing. We present SCFS, a cloud-backed file system that addresses these issues and provides strong consistency and near-POSIX semantics on top of eventually-consistent cloud storage services. SCFS provides a pluggable backplane that allows it to work with various storage clouds or a cloud-of-clouds (for added dependability). It also exploits some design opportunities inherent in the current cloud services through a set of novel ideas for cloud-backed file systems: always write and avoid reading, modular coordination, private name spaces and consistency anchors.

1 Introduction

File backup, data archival and collaboration are among the top usages of the cloud in companies [1], and they are normally based on cloud storage services like the Amazon S3, Dropbox, Google Drive and Microsoft SkyDrive. These services are popular because of their ubiquitous accessibility, pay-as-you-go model, high scalability, and ease of use. A cloud storage service can be accessed in a convenient way with a client application that interfaces the local file system and the cloud. Such services can be broadly grouped in two classes: (1) personal file synchronization services (e.g., DropBox) and (2) cloud-backed file systems (e.g., S3FS [6]).

Services of the first class – personal file synchronization – are usually composed of a back-end storage cloud and a client application that interacts with the local file system through a monitoring interface like *inotify* (in Linux). Recent works show that this interaction model can lead to reliability and consistency problems on the stored data [41], as well as CPU and bandwidth over usage under certain workloads [35]. In particular, given the fact that these monitoring components lack an understanding of when data or metadata is made persistent in the local storage, this can lead to corrupted data being saved in the cloud. A possible solution to these difficulties would be to modify the file system to increase the integration between client application and local storage.

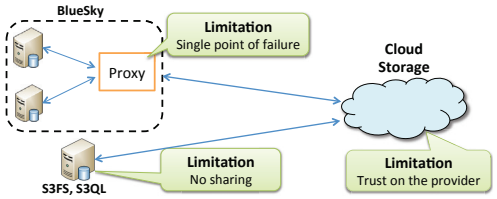


Figure 1: Cloud-backed file systems and their limitations.

The second class of services – cloud-backed file systems – solves the problem in a more generic way. This approach is typically implemented at user-level, following one of the two architectural models represented in Figure 1. The first model is shown at the top of the figure and is followed by BlueSky [39] and several commercial storage gateways. In this model, a proxy component is placed in the network infrastructure of the organization, acting as a file server to multiple clients and supporting access protocols such as NFS and CIFS. The proxy implements the core file system functionality and calls the cloud to store and retrieve files. The main limitations are that the proxy can become a performance bottleneck and a single point of failure. Moreover, in BlueSky (and some other systems) there is no coordination between different proxies accessing the same files. The second model is implemented by open-source solutions like S3FS [6] and S3QL [7] (bottom of Figure 1). In this model, clients access the cloud directly, without the interposition of a proxy. Consequently, there is no longer a single point of failure, but the model misses the convenient rendezvous point for synchronization, making it harder to support controlled file sharing among clients.

A common limitation of the two classes of services is the need to trust the cloud provider with respect to the stored data confidentiality, integrity and availability. Although confidentiality can be guaranteed by making clients (or the proxy) encrypt files before sending them to the cloud, sharing encrypted files requires a key distribution mechanism, which is not easy to implement in this environment. Integrity is provided by systems like SUNDR [34], which requires the execution of specific code on the cloud provider, currently not possible when using unmodified storage services. Availability against cloud failures to the best of our knowledge is not provided by any of the current cloud-backed file systems.

This paper presents the *Shared Cloud-backed File System (SCFS)*,[1] a storage solution that addresses the aforementioned limitations. SCFS allows entities to *share* files in a secure and fault-tolerant way, improving the durability guarantees. It also ensures strong consistency on file accesses, and provides a backplane that can plug on multiple different cloud storage services.

SCFS leverages almost 30 years of distributed file systems research, integrating classical ideas like consistency-on-close semantics [28] and separation of data and metadata [21], with recent trends such as using cloud services as (unmodified) storage backends [20, 39] and increasing dependability by resorting to multiple clouds [9, 12, 15]. These ideas were augmented with the following *novel techniques* for cloud-backed storage design:

- *Always write / avoid reading:* SCFS always pushes updates of file contents to the cloud (besides storing them locally), but resolves reads locally whenever possible. This mechanism has a positive impact in the reading latency. Moreover, it reduces costs because writing to the cloud is typically cheap, on the contrary of reading that tends to be expensive.[2]
- *Modular coordination:* SCFS uses a fault-tolerant coordination service, instead of an embedded lock and metadata manager, as most distributed file systems [10, 32, 40]. This service has the benefit of assisting the management of consistency and sharing. Moreover, the associated modularity is important to support different fault tolerance tradeoffs.
- *Private Name Spaces:* SCFS uses a new data structure to store metadata information about files that are not shared between users (which is expected to be the majority [33]) as a single object in the storage cloud. This relieves the coordination service from maintaining information about such private files and improves the performance of the system.
- *Consistency anchors:* SCFS employs this novel mechanism to achieve strong consistency, instead of the eventual consistency [38] offered by most cloud storage services, a model typically considered unnatural by a majority of programmers. This mechanism provides a familiar abstraction – a file system – without requiring modifications to cloud services.
- *Multiple redundant cloud backends:* SCFS may employ a cloud-of-clouds backplane [15], making the system tolerant to data corruption and unavailability of cloud providers. All data stored in the clouds is encrypted for confidentiality and encoded for storage-efficiency.

[1] SCFS is available at http://code.google.com/p/depsky/wiki/SCFS.

[2] For example, in Amazon S3, writing is free, but reading a GB is more expensive ($0.12 after the first GB/month) than storing data during a month ($0.09 per GB). Google Cloud Storage's prices are similar.

The use case scenarios of SCFS include both individuals and large organizations, which are willing to explore the benefits of cloud-backed storage (optionally, with a cloud-of-clouds backend). For example: *a secure personal file system* – similar to Dropbox, iClouds or SkyDrive, but without requiring complete trust on any single provider; *a shared file system for organizations* – cost-effective storage, but maintaining control and confidentiality of the organizations' data; *an automatic disaster recovery system* – the files are stored by SCFS in a cloud-of-clouds backend to survive disasters not only in the local IT systems but also of individual cloud providers; *a collaboration infrastructure* – dependable data-based collaborative applications without running code in the cloud, made easy by the POSIX-like API for sharing files.

Despite the fact that distributed file systems are a well-studied subject, our work relates to an area where further investigation is required – cloud-backed file systems – and where the practice is still somewhat immature. In this sense, besides presenting a system that explores a novel region of the cloud storage design space, the paper contributes with a set of generic principles for cloud-backed file system design, reusable in further systems with different purposes than ours.

2 SCFS Design

2.1 Design Principles

This section presents a set of design principles that are followed in SCFS:

Pay-per-ownership. Ideally, a shared cloud-backed file system should charge each owner of an account for the files it creates in the service. This principle is important because it leads to a flexible usage model, e.g., allowing different organizations to share directories paying only for the files they create. SCFS implements this principle by reusing the protection and isolation between different accounts granted by the cloud providers (see §2.6).

Strong consistency. A file system is a more familiar storage abstraction to programmers than the typical basic interfaces (e.g., REST-based) given by cloud storage services. However, to emulate the semantics of a POSIX file system, strong consistency has to be provided. SCFS follows this principle by applying the concept of consistency anchors (see §2.4). Nevertheless, SCFS optionally supports weaker consistency.

Service-agnosticism. A cloud-backed file system should rule out from its design any feature that is not supported by the backend cloud(s). The importance of this principle derives from the difficulty (or impossibility) in obtaining modifications of the service of the best-of-breed commercial clouds. Accordingly, SCFS does not assume any special feature of storage clouds besides on-demand access to storage and basic access control lists.

Multi-versioning. A shared cloud-backed file system should be able to store several versions of the files for error recovery [23]. An important advantage of having a cloud as backend is its potentially unlimited capacity and scalability. SCFS keeps old versions of files and deleted files until they are definitively removed by a configurable garbage collector.

2.2 Goals

A primary goal of SCFS is to allow clients to share files in a controlled way, providing the necessary mechanisms to guarantee security (integrity and confidentiality; availability despite cloud failures is optional). An equally important goal is to increase data durability by exploiting the resources granted by storage clouds and keeping several versions of files.

SCFS also aims to offer a natural file system API with strong consistency. More specifically, SCFS supports consistency-on-close semantics [28], guaranteeing that when a file is closed by a user, all updates it saw or did are observable by the rest of the users. Since most storage clouds provide only eventual consistency, we resort to a coordination service [13, 29] for maintaining file system metadata and synchronization.

A last goal is to leverage the clouds' services scalability, supporting large numbers of users and files as well as large data volumes. However, SCFS is not intended to be a big-data file system, since file data is uploaded to and downloaded from one or more clouds. On the contrary, a common principle for big-data processing is to take computation to the data (e.g., MapReduce systems).

2.3 Architecture Overview

Figure 2 represents the SCFS architecture with its three main components: the *backend cloud storage* for maintaining the file data (shown as a cloud-of-clouds, but a single cloud can be used); the *coordination service* for managing the metadata and to support synchronization; and the *SCFS Agent* that implements most of the SCFS functionality, and corresponds to the file system client mounted at the user machine.

The separation of file data and metadata has been often used to allow parallel access to files in parallel file systems (e.g., [21, 40]). In SCFS we take this concept further and apply it to a cloud-backed file system. The fact that a distinct service is used for storing metadata gives flexibility, as it can be deployed in different ways depending on the users needs. For instance, our general architecture assumes that metadata is kept in the cloud, but a large organization could distribute the metadata service over its own sites for disaster tolerance.

Metadata in SCFS is stored in a coordination service. Three important reasons led us to select this approach instead of, for example, a NoSQL database or some custom service (as in other file systems). First, coordination

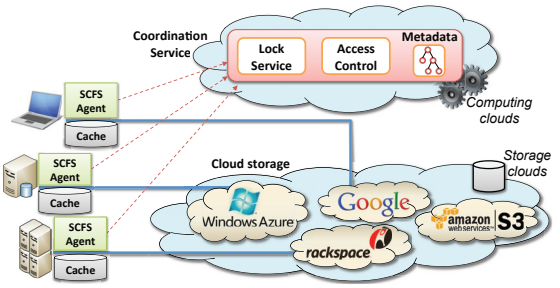


Figure 2: SCFS architecture.

services offer *consistent* storage with enough capacity for this kind of data, and thus can be used as *consistency anchors* for cloud storage services (see next section). Second, coordination services implement complex replication protocols to ensure *fault tolerance* for metadata storage. Finally, these systems support operations with *synchronization* power [26] that can be used to implement fundamental file system functionalities, such as locking.

File data is maintained both in the storage cloud and locally in a cache at the client machine. This strategy is interesting in terms of performance, costs and availability. As cloud accesses usually entail large latencies, SCFS attempts to keep a copy of the accessed files in the user's machine. Therefore, if the file is not modified by another client, subsequent reads do not need to fetch the data from the clouds. As a side effect, there are cost savings as there is no need to pay to download the file. On the other hand, we follow the approach of writing everything to the cloud, as most providers let clients upload files for free as an incentive to use their services. Consequently, no completed update is lost in case of a local failure.

According to our design, the storage cloud(s) and the coordination service are external services. SCFS can use any implementation of such services as long as they are compatible (provide compliant interfaces, access control and the required consistency). We will focus the rest of this section on the description of the SCFS Agent and its operation principles, starting with how it implements consistent storage using weakly consistent storage clouds.

2.4 Strengthening Cloud Consistency

A key innovation of SCFS is the ability to provide strongly consistent storage over the eventually-consistent services offered by clouds [38]. Given the recent interest in strengthening eventual consistency in other areas, we describe the general technique here, decoupled from the file system design. A complete formalization and correctness proof of this technique is presented in a companion technical report [14].

The approach uses two storage systems, one with limited capacity for maintaining metadata and another to save the data itself. We call the metadata store a *consistency anchor* (CA) and require it to enforce some desired consistency guarantee S (e.g., linearizability [27]), while the

WRITE(id, v):	READ(id):
w1: $h \leftarrow \mathsf{Hash}(v)$	**r1:** $h \leftarrow$ CA.read(id)
w2: SS.write($id\|h, v$)	**r2:** **do** $v \leftarrow$ SS.read($id\|h$) **while** $v = null$
w3: CA.write(id, h)	**r3:** **return** $(\mathsf{Hash}(v) = h)?v : null$

Figure 3: Algorithm for increasing the consistency of the storage service (SS) using a consistency anchor (CA).

storage service (SS) may only offer eventual consistency. The objective is to provide a composite storage system that satisfies S, even if the data is kept in SS.

The algorithm for improving consistency is presented in Figure 3, and the insight is to anchor the consistency of the resulting storage service on the consistency offered by the CA. For writing, the client starts by calculating a collision-resistant hash of the data object (step w1), and then saves the data in the SS together with its identifier id concatenated with the hash (step w2). Finally, data's identifier and hash are stored in the CA (step w3). Every write operation creates a new version of the data object and garbage collection is required to reclaim the storage space of no longer needed versions.

For reading, the client has to obtain the current hash of the data from CA (step r1), and then needs to keep on fetching the data object from the SS until a copy is available (step r2). The loop is necessary due to the eventual consistency of the SS – after a write completes, the new hash can be immediately acquired from the CA, but the data is only eventually available in the SS.

2.5 SCFS Agent

2.5.1 Local Services

The design of the SCFS Agent is based on the use of three *local* services that abstract the access to the coordination service and the storage cloud backend.

Storage service. The storage service provides an interface to save and retrieve variable-sized objects from the cloud storage. SCFS overall performance is heavily affected by the latency of remote (Internet) cloud accesses. To address this problem, we read and write whole files as objects in the cloud, instead of splitting them in blocks and accessing block by block. This allows most of the client files (if not all) to be stored locally, and makes the design of SCFS simpler and more efficient for small-to-medium sized files.

To achieve adequate performance, we rely on two levels of cache, whose organization has to be managed with care in order to avoid impairing consistency. First, all files read and written are copied locally, making the local disk a large and long-term cache. More specifically, the disk is seen as an LRU file cache with GBs of space, whose content is validated in the coordination service before being returned, to ensure that the most recent version of the file is used. Second, a main memory LRU cache (hundreds of MBs) is employed for holding open files. This is aligned with our consistency-on-close semantics, since, when the file is closed, all updated metadata and data kept in memory are flushed to the local disk and the clouds.

Actual data transfers between the various storage locations (memory, disk, clouds) are defined by the durability levels required by each type of system call. Table 1 shows examples of POSIX calls that cause data to be stored at different levels, together with their location, storage latency and fault tolerance. For instance, a write in an open file is stored in the memory cache, which gives no durability guarantees (Level 0). Calling `fsync` flushes the file (if modified) to the local disk, achieving the standard durability of local file systems, i.e., against process or system crashes (Level 1). When a file is closed, it is eventually written to the cloud. A system backed by a single cloud provider can survive a local disk failure but not a cloud provider failure (Level 2). However, in SCFS with a cloud-of-clouds backend, files are written to a set of clouds, such that failure of up to f providers is tolerated (Level 3), being f a system parameter (see §3.2).

Level	*Location*	*Latency*	*Fault tol.*	*Sys. call*
0	main memory	microsec	none	write
1	local disk	millisec	crash	fsync
2	cloud	seconds	local disk	close
3	cloud-of-clouds[1]	seconds	f clouds	close

Table 1: SCFS durability levels and the corresponding data location, write latency, fault tolerance and example system calls. [1]Supported by SCFS with the cloud-of-clouds backend.

Metadata service. The metadata service resorts to the coordination service to store file and directory metadata, together with information required for enforcing access control. Each file system object is represented in the coordination service by a metadata tuple containing: the object name, the type (file, directory or link), its parent object (in the hierarchical file namespace), the object metadata (size, date of creation, owner, ACLs, etc.), an opaque identifier referencing the file in the storage service (and, consequently, in the storage cloud) and the collision-resistant hash (SHA-1) of the contents of the current version of the file. These two last fields represent the id and the $hash$ stored in the consistency anchor (see §2.4). Metadata tuples are accessed through a set of operations offered by the local metadata service, which are then translated into different calls to the coordination service.

Most application actions and system call invocations are translated to several metadata accesses at the file system level (e.g., opening a file with the `vim` editor can cause more than five `stat` calls for the file). To deal with these access bursts, a small short-term metadata cache is kept in main memory (up to few MBs for tens of milliseconds). The objective of this cache is to reuse the data fetched from the coordination service for at least the amount of time spent to obtain it from the network.

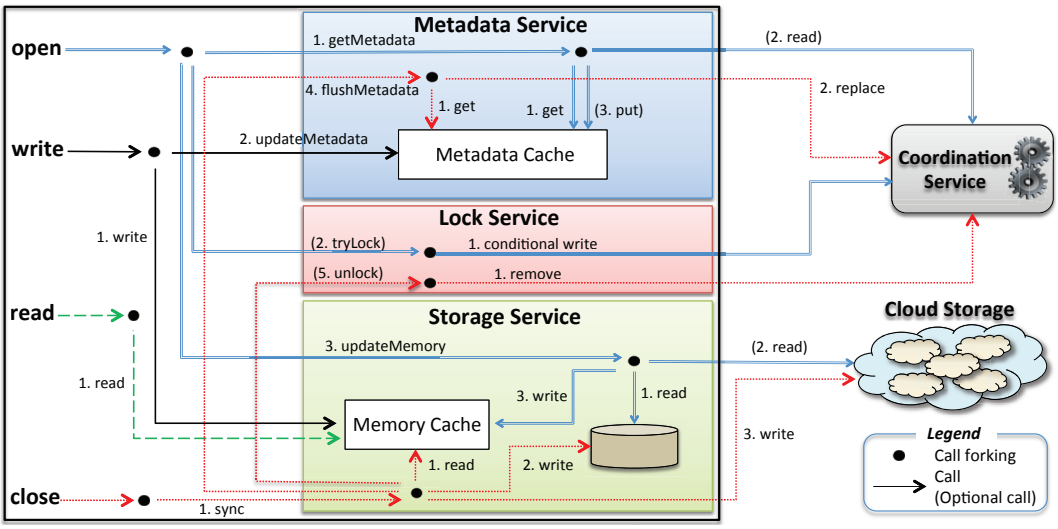


Figure 4: Common file system operations in SCFS. The following conventions are used: 1) at each call forking (the dots between arrows), the numbers indicate the order of execution of the operations; 2) operations between brackets are optional; 3) each file system operation (e.g., open/close) has a different line pattern.

Notice that accessing cached metadata can lead to violations of strong consistency. For this reason, we maintain such cached information for very short time periods, only to serve the file system calls originated from the same high-level action over a file (e.g., opening or saving a document). In §4.4 we show that this cache significantly boosts the performance of the system.

Locking service. As in most consistent file systems, we use *locks* to avoid write-write conflicts. The lock service is basically a wrapper for implementing coordination recipes for locking using the coordination service of choice [13, 29]. The only strict requirement is that the lock entries inserted are ephemeral. In practice, locks can be represented by ephemeral znodes in Zookeeper or timed tuples in DepSpace, ensuring they will disappear (automatically unlocking the file) in case a SCFS client that locked a file crashes before uploading its updates and releasing the lock (see next section).

Opening a file for reading does not require locking it. Read-write conflicts are automatically addressed when uploading and downloading whole files and using consistency anchors (see §2.4) which ensure the most recent version of file (according to consistency-on-close) will be read upon its opening.

2.5.2 File Operations

Figure 4 illustrates the execution of the four main file system calls (open, write, read and close) in SCFS.

Opening a file. The tension between provisioning strong consistency and suffering high latency in cloud access led us to provide consistency-on-close semantics [28] and synchronize files only in the open and close operations. Moreover, given our aim of having most client files (if not all) locally stored, we opted for reading and writing whole files from the cloud. With this in mind, the open operation comprises three main steps: (i) read the file metadata, (ii) optionally create a lock if the file is opened for writing, and (iii) read the file data to the local cache. Notice that these steps correspond to an implementation of the *read* algorithm of Figure 3, with an extra step to ensure exclusive access to the file for writing.

Reading the metadata entails fetching the file metadata from the coordination service, if it is not available in the metadata cache, and then make an update to this cache. Locking the file is necessary to avoid write-write conflicts, and if it fails, an error is returned. File reads are either done in the local cache (memory or disk) or in the cloud. The local file version (if available) is compared with the version stored in the metadata service. If a newer version exists, it is read from the cloud and cached in the local disk and in main memory. If there is no space for the file in main memory (e.g., there are too many open files), the data of the least recently used file is first pushed to disk (as a cache extension) to release space.

Write and read. These two operations only need to interact with the local storage. Writing to a file requires updating the memory-cached file and the associated metadata cache entry (e.g., the size and the last-modified timestamp). Reading just causes the data to be fetched from the main memory cache (as it was copied there when the file was opened).

Closing a file. Closing a file involves the synchronization of cached data and metadata with the coordination service and the cloud storage. First, the updated file data is copied to the local disk and to the storage cloud. Then, if the cached metadata was modified, it is pushed to the coordination service. Lastly, the file is unlocked if it was originally opened for writing. Notice that these steps correspond to the *write* algorithm of Figure 3.

As expected, if the file was not modified since opened or was opened in read-only mode, no synchronization is required. From the point of view of consistency and durability, a write to the file is complete only when the file is closed, respecting the consistency-on-close semantics.

2.5.3 Garbage Collection

During normal operation, SCFS saves new file versions without deleting the previous ones, and files removed by the user are just marked as deleted in the associated metadata. These two features support the recovery of old versions of the files, which is useful for some applications. Keeping old versions of files increases storage costs, and therefore, SCFS includes a flexible garbage collector to enable various policies for reclaiming space.

Garbage collection runs in isolation at each SCFS Agent, and the decision about reclaiming space is based on the preferences (and budgets) of individual users. By default, its activation is guided by two parameters defined upon the mounting of the file system: *number of written bytes* W and *number of versions to keep* V. Every time an SCFS Agent writes more than W bytes, it starts the garbage collector as a separated thread that runs in parallel with the rest of the system (other policies are possible). This thread fetches the list of files owned by this user and reads the associated metadata from the coordination service. Next, it issues commands to delete old file data versions from the cloud storage, such that only the last V versions are kept (refined policies that keep one version per day or week are also possible). Additionally, it also eliminates the versions removed by the user. Later on, the corresponding metadata entries are also erased from the coordination service.

2.6 Security Model

The security of a shared cloud storage system is a tricky issue, as the system is constrained by the access control capabilities of the backend clouds. A straw-man implementation would allow all clients to use the same account and privileges on the cloud services, but this has two drawbacks. First, any client would be able to modify or delete all files, making the system vulnerable to malicious users. Second, a single account would be charged for all clients, preventing the pay-per-ownership model.

Instead of classical Unix modes (*owner*, *group*, *others*; *read*, *write*, *execute*), SCFS implements ACLs [22]. The owner O of a file can give access permissions to another user U through the `setfacl` call, passing as parameters file name, identifier of user U, and permissions. Similarly, `getfacl` retrieves the permissions of a file.

As a user has separate accounts in the various cloud providers, and since each probably has a different identifier, SCFS needs to associate with every client a list of cloud canonical identifiers. This association is kept in a tuple in the coordination service, and is loaded when the client mounts the file system for the first time. When the SCFS Agent intercepts a `setfacl` request from a client O to set permissions on a file for a user U, the following steps are executed: (i) the agent uses the two lists of cloud canonical identifiers (of O and U) to update the ACLs of the objects that store the file data in the clouds with the new permissions; and then, (ii) it also updates the ACL associated with the metadata tuple of the file in the coordination service to reflect the new permissions.

Notice that we do not trust the SCFS Agent to implement the access control verification, since it can be compromised by a malicious user. Instead, we rely on the access control enforcement of the coordination service and the cloud storage.

2.7 Private Name Spaces

One of the goals of SCFS is to scale in terms of users and files. However, the use of a coordination service (or any centralized service) could potentially create a scalability bottleneck, as this kind of service normally maintains all data in main memory [13, 29] and requires a distributed agreement to update the state of the replicas in a consistent way. To address this problem, we take advantage of the observation that, although file sharing is an important feature of cloud-backed storage systems, the majority of the files are not shared between different users [20, 33]. Looking at the SCFS design, all files and directories that are not shared (and thus not visible to other users) do not require a specific entry in the coordination service, and instead can have their metadata grouped in a single object saved in the cloud storage. This object is represented by a *Private Name Space* (PNS).

A PNS is a local data structure kept by the SCFS Agent's metadata service, containing the metadata of all private files of a user. Each PNS has an associated PNS tuple in the coordination service, which contains the user name and a reference to an object in the cloud storage. This object keeps a copy of the serialized metadata of all private files of the user.

Working with non-shared files is slightly different from what was shown in Figure 4. When mounting the file system, the agent fetches the user's PNS entry from the coordination service and the metadata from the cloud storage, locking the PNS to avoid inconsistencies caused by two clients logged in as the same user. When opening a file, the user gets the metadata locally as if it was in cache (since the file is not shared), and if needed fetches data from the cloud storage (as in the normal case). On close, if the file was modified, both data and metadata are updated in the cloud storage. The close operation completes when both updates finish.

When permissions change in a file, its metadata can be removed (resp. added) from a PNS, causing the creation (resp. removal) of the corresponding metadata tuple in the coordination service.

With PNSs, the amount of storage used in the coordination service is proportional to the percentage of shared files in the system. Previous work show traces with 1 million files where only 5% of them are shared [33]. Without PNSs, the metadata for these files would require 1 million tuples of around 1KB, for a total size of 1GB of storage (the approximate size of a metadata tuple is 1KB, assuming 100B file names). With PNSs, only 50 thousand tuples plus one PNS tuple per user would be needed, requiring a little more than 50MB of storage. Even more importantly, by resorting to PNSs, it is possible to reduce substantially the number of accesses to the coordination service, allowing more users and files to be served.

3 SCFS Implementation

SCFS is implemented in Linux as a user-space file system based on FUSE-J, which is a wrapper to connect the SCFS Agent to the FUSE library. Overall, the SCFS implementation comprises 6K lines of commented Java code, excluding any coordination service or storage backend code. We opted to develop SCFS in Java mainly because most of the backend code (the coordination and storage services) were written in Java and the high latency of cloud accesses make the overhead of using a Java-based file system comparatively negligible.

3.1 Modes of Operation

Our implementation of SCFS supports three modes of operation, based on the consistency and sharing requirements of the stored data.

The first mode, *blocking*, is the one described up to this point. The second mode, *non-blocking*, is a weaker version of SCFS in which closing a file does not block until the file data is on the clouds, but only until it is written locally and enqueued to be sent to the clouds in background. In this model, the file metadata is updated and the associated lock released only after the file contents are updated to the clouds, and not when the close call returns (so mutual exclusion is preserved). Naturally, this model leads to a significant performance improvement at cost of a reduction of the durability and consistency guarantees. Finally, the *non-sharing* mode is interesting for users that do not need to share files, and represents a design similar to S3QL [7], but with the possibility of using a cloud-of-clouds instead of a single storage service. This version does not require the use of the coordination service, and all metadata is saved on a PNS.

3.2 Backends

SCFS can be plugged to several backends, including different coordination and cloud storage services. This paper focuses on the two backends of Figure 5. The first one is based on Amazon Web Services (AWS), with an EC2 VM running the coordination service and file data being stored in S3. The second backend makes use of the cloud-of-clouds (CoC) technology, recently shown to be practical [9, 12, 15]. A distinct advantage of the CoC backend is that it removes any dependence of a single cloud provider, relying instead on a quorum of providers. It means that data security is ensured even if f out-of $3f+1$ of the cloud providers suffer *arbitrary faults*, which encompasses unavailability and data deletion, corruption or creation [15]. Although cloud providers have their means to ensure the dependability of their services, the recurring occurrence of outages, security incidents (with internal or external origins) and data corruptions [19, 24] justifies the need for this sort of backend in several scenarios.

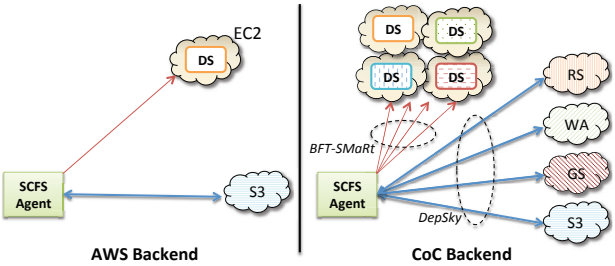


Figure 5: SCFS with Amazon Web Services (AWS) and Cloud-of-Clouds (CoC) backends.

Coordination services. The current SCFS prototype supports two coordination services: Zookeeper [29] and DepSpace [13] (in particular, its durable version [16]). These services are integrated at the SCFS Agent with simple wrappers, as both support storage of small data entries and can be used for locking. Moreover, these coordination services can be deployed in a replicated way for fault tolerance. Zookeeper requires $2f+1$ replicas to tolerate f crashes through the use of a Paxos-like protocol [30] while DepSpace uses either $3f+1$ replicas to tolerate f arbitrary/Byzantine faults or $2f+1$ to tolerate crashes (like Zookeeper), using the BFT-SMaRt replication engine [17]. Due to the lack of hierarchical data structures in DepSpace, we had to extend it with support for triggers to efficiently implement file system operations like rename.

Cloud storage services. SCFS currently supports Amazon S3, Windows Azure Blob, Google Cloud Storage, Rackspace Cloud Files and all of them forming a cloud-of-clouds backend. The implementation of single-cloud backends is simple: we employ the Java library made available by the providers, which accesses the cloud storage service using a REST API over SSL. To implement the cloud-of-clouds backend, we resort to an extended version of DepSky [15] that supports a new operation, which instead of reading the last version of a data unit, reads the version with a given hash, if available (to implement the consistency anchor algorithm - see §2.4). The hashes of all versions of the data are stored in DepSky's internal metadata object, stored in the clouds.

Figure 6 shows how a file is securely stored in the cloud-of-clouds backend of SCFS using DepSky (see [15] for details). The procedure works as follows: (1) a random key K is generated, (2) this key is used to encrypt the file and (3) the encrypted file is encoded and each block is stored in different clouds together with (4) a share of K, obtained through secret sharing. Stored data security (confidentiality, integrity and availability) is ensured by the fact that no single cloud alone has access to the data since K can only be recovered with two or more shares and that quorum reasoning is applied to discover the last version written. In the example of the figure, where a single faulty cloud is tolerated, two clouds need to be accessed to recover the file data.

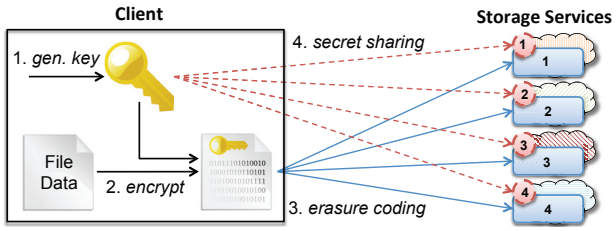


Figure 6: A write in SCFS using the DepSky protocols.

4 Evaluation

This section evaluates SCFS using AWS and CoC backends, operating in different modes, and comparing them with other cloud-backed file systems. The main objective is to understand how SCFS behaves with some representative workloads and to shed light on the costs of our design.

4.1 Setup & Methodology

Our setup considers a set of clients running on a cluster of Linux 2.6 machines with two quad-core 2.27 GHz Intel Xeon E5520, 32 GB of RAM and a 15K RPM SCSI HD. This cluster is located in Portugal.

For SCFS-AWS (Figure 5, left), we use Amazon S3 (US) as a cloud storage service and a single EC2 instance hosted in Ireland to run DepSpace. For SCFS-CoC, we use DepSky with four storage providers and run replicas of DepSpace in four computing cloud providers, tolerating a single fault both in the storage service and in the coordination service. The storage clouds were Amazon S3 (US), Google Cloud Storage (US), Rackspace Cloud Files (UK) and Windows Azure (UK). The computing clouds were EC2 (Ireland), Rackspace (UK), Windows Azure (Europe) and Elastichosts (UK). In all cases, the VM instances used were EC2 M1 Large [2] (or similar).

The evaluation is based on a set of benchmarks following recent recommendations [37], all of them from *Filebench* [3]. Moreover, we created two new benchmarks to simulate some behaviors of interest for cloud-backed file systems.

We compare six SCFS variants considering different modes of operation and backends (see Table 2) with two popular open source S3-backed files systems: S3QL [7] and S3FS [6]. Moreover, we use a FUSE-J-based local file system (LocalFS) implemented in Java as a baseline to ensure a fair comparison, since a native file system presents much better performance than a FUSE-J file system. In all SCFS variants, the metadata cache expiration time was set to 500 ms and no private name spaces were used. Alternative configurations are evaluated in §4.4.

	Blocking	*Non-blocking*	*Non-sharing*
AWS	SCFS-AWS-B	SCFS-AWS-NB	SCFS-AWS-NS
CoC	SCFS-CoC-B	SCFS-CoC-NB	SCFS-CoC-NS

Table 2: SCFS variants with different modes and backends.

4.2 Micro-benchmarks

We start with six Filebench micro-benchmarks [3]: sequential reads, sequential writes, random reads, random writes, create files and copy files. The first four benchmarks are IO-intensive and do not consider open, sync or close operations, while the last two are metadata-intensive. Table 3 shows the results for all considered file systems.

The results for sequential and random reads and writes show that the behavior of the evaluated file systems is similar, with the exception of S3FS and S3QL. The low performance of S3FS comes from its lack of main memory cache for opened files [6], while S3QL's low random write performance is the result of a known issue with FUSE that makes small chunk writes very slow [8]. This benchmark performs 4KB-writes, much smaller than the recommended chunk size for S3QL, 128KB.

The results for create and copy files show a difference of three to four orders of magnitude between the local or single-user cloud-backed file system (SCFS-*-NS, S3QL and LocalFS) and a shared or blocking cloud-backed file system (SCFS-*-NB, SCFS-*-B and S3FS). This is not surprising, given that SCFS-*-{NB,B} access the coordination service in each create, open or close operation. Similarly, S3FS accesses S3 in each of these operations, being even slower. Furthermore, the latencies of SCFS-*-NB variants are dominated by the coordination service access (between 60-100 ms per access), while in the SCFS-*-B variants such latency is dominated by read and write operations in the cloud storage.

4.3 Application-based Benchmarks

In this section we present two application-based benchmarks for potential uses of cloud-backed file systems.

File Synchronization Service. A representative workload for SCFS corresponds to its use as a personal file synchronization service [20] in which desktop application files (spreadsheets, documents, presentations, etc.) are stored and shared. A new benchmark was designed to simulate opening, saving and closing a text document with OpenOffice Writer.

Micro-benchmark	#Operations	File size	SCFS-AWS			SCFS-CoC			S3FS	S3QL	LocalFS
			NS	NB	B	NS	NB	B			
sequential read	1	4MB	1	1	1	1	1	1	6	1	1
sequential write	1	4MB	1	1	1	1	1	1	2	1	1
random 4KB-read	256k	4MB	11	11	15	11	11	11	15	11	11
random 4KB-write	256k	4MB	35	39	39	35	35	36	52	152	37
create files	200	16KB	1	102	229	1	95	321	596	1	1
copy files	100	16KB	1	137	196	1	94	478	444	1	1

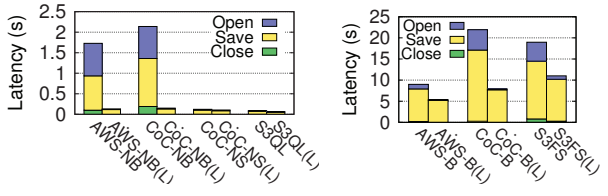
Table 3: Latency of several Filebench micro-benchmarks for SCFS (six variants), S3QL, S3FS and LocalFS (in seconds).

The benchmark follows the behavior observed in traces of a real system, which are similar to other modern desktop applications [25]. Typically, the files managed by the cloud-backed file system are just copied to a temporary directory on the local file system where they are manipulated as described in [25]. Nonetheless, as can be seen in the benchmark definition (Figure 7), these actions (especially save) still impose a lot of work on the file system.

Open Action: 1 open(f,rw), 2 read(f), 3-5 open-write-close(lf1), 6-8 open-read-close(f), 9-11 open-read-close(lf1)
Save Action: 1-3 open-read-close(f), 4 close(f), 5-7 open-read-close(lf1), 8 delete(lf1), 9-11 open-write-close(lf2), 12-14 open-read-close(lf2), 15 truncate(f,0), 16-18 open-write-close(f), 19-21 open-fsync-close(f), 22-24 open-read-close(f), 25 open(f,rw)
Close Action: 1 close(f), 2-4 open-read-close(lf2), 5 delete(lf2)

Figure 7: File system operations invoked in the file synchronization benchmark, simulating an OpenOffice document open, save and close actions (*f* is the odt file and *lf* is a lock file).

Figure 8 shows the average latency of each of the three actions of our benchmark for SCFS, S3QL and S3FS, considering a file of 1.2MB, which corresponds to the average file size observed in 2004 (189KB) scaled-up 15% per year to reach the expected value for 2013 [11].



(a) Non-blocking. (b) Blocking.

Figure 8: Latency of file synchronization benchmark actions (see Figure 7) with a file of 1.2MB. The (L) variants maintain lock files in the local file system. All labels starting with CoC or AWS represent SCFS variants.

Figure 8(a) shows that SCFS-CoC-NS and S3QL exhibit the best performance among the evaluated file systems, having latencies similar to a local file system (where a save takes around 100 ms). This shows that the added dependability of a cloud-of-clouds storage backend does not prevent a cloud-backed file system to behave similarly to a local file system, if the correct design is employed.

Our results show that SCFS-*-NB requires substantially more time for each phase due to the number of accesses to the coordination service, especially to deal with the lock files used in this workload. Nonetheless, saving a file in this system takes around 1.2 s, which is acceptable from the usability point of view. A much slower behavior is observed in the SCFS-*-B variants, where the creation of a lock file makes the system block waiting for this small file to be pushed to the clouds.

We observed that most of the latency comes from the manipulation of lock files. However, the files accessed did not need to be stored in the SCFS partition, since the locking service already prevents write-write conflicts between concurrent clients. We modified the benchmark to represent an application that writes lock files locally (in `/tmp`), just to avoid conflicts between applications in the same machine. The (L) variants in Figure 8 represent results with such local lock files. These results show that removing the lock files makes the cloud-backed system much more responsive. The takeaway here is that the usability of blocking cloud-backed file systems could be substantially improved if applications take into consideration the limitations of accessing remote services.

Sharing files. Personal cloud storage services are often used for sharing files in a controlled and convenient way [20]. We designed an experiment for comparing the time it takes for a shared file written by a client to be available for reading by another client, using SCFS-*-{NB,B}. We did the same experiment considering a Dropbox shared folder (creating random files to avoid deduplication). We acknowledge that the Dropbox design [20] is quite different from SCFS, but we think it is illustrative to show how a cloud-backed file system compares with a popular file synchronization service.

The experiment considers two clients A and B deployed in our cluster. We measured the elapsed time between the instant client A closes a variable-size file that it wrote to a shared folder and the instant it receives an UDP ACK from client B informing the file was available. Clients A and B are Java programs running in the same LAN, with a ping latency of around 0.2 ms, which is negligible considering the latencies of reading and writing. Figure 9 shows the results of this experiment for different file sizes.

The results show that the latency of sharing in SCFS-*-B is much smaller than what people experience in current personal storage services. These results do not consider

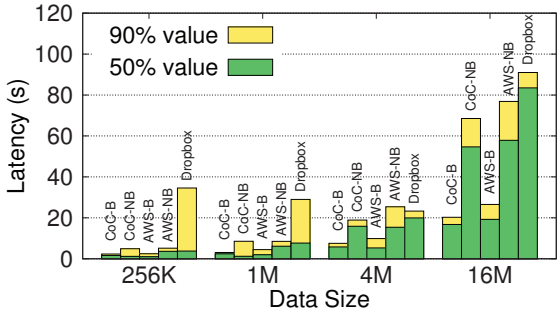


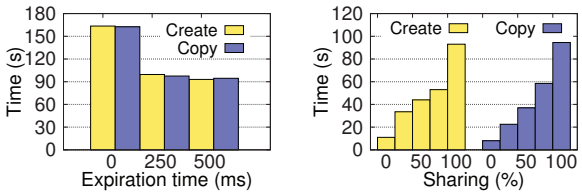
Figure 9: Sharing file 50th and 90th latency for SCFS (CoC B and NB, AWS B and NB) and Dropbox for different file sizes.

the benefits of deduplication, which SCFS currently does not support. However, if a user encrypts its critical files locally before storing them in Dropbox, the effectiveness of deduplication will be decreased significantly.

Figure 9 also shows that the latency of the blocking SCFS is much smaller than the non-blocking version with both AWS and CoC backends. This is explained by the fact that the SCFS-*-B waits for the file write to complete before returning to the application, making the benchmark measure only the delay of reading the file. This illustrates the benefits of SCFS-*-B: when A completes its file closing, it knows the data is available to any other client the file is shared with. We think this design can open interesting options for collaborative applications based on SCFS.

4.4 Varying SCFS Parameters

Figure 10 shows some results for two metadata-intensive micro-benchmarks (copy and create files) for SCFS-CoC-NB with different metadata cache expiration times and percentages of files in private name spaces.



(a) Metadata cache. (b) Private Name Spaces.

Figure 10: Effect of metadata cache expiration time and PNSs with different file sharing percentages in two metadata intensive micro-benchmarks.

As described in §2.5.1, we implemented a short-lived metadata cache to deal with bursts of metadata access operations (e.g., `stat`). All previous experiments used an expiration time of 500 ms for this cache. Figure 10(a) shows how changing this value affects the performance of the system. The results clearly indicate that not using such metadata cache (expiration time equals zero) severely degrades the system performance. However, beyond some point, increasing it does not bring much benefit either.

Figure 10(b) displays the latency of the same benchmarks considering the use of PNS (see §2.7) with different percentages of files shared between more than one user. Recall that all previous results consider full-sharing (100%), without using PNS, which is a worst case scenario. As expected, the results show that as the number of private files increases, the performance of the system improves. For instance, when only 25% of the files are shared – more than what was observed in the most recent study we are aware of [33] – the latency of the benchmarks decreases by a factor of roughly 2.5 (create files) and 3.5 (copy files).

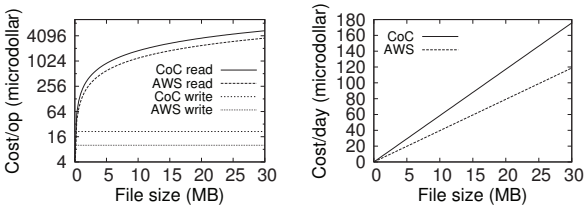
4.5 SCFS Operation Costs

Figure 11 shows the costs associated with operating and using SCFS. The *fixed operation costs* of SCFS comprise mainly the maintenance of the coordination service running in one or more VMs deployed in cloud providers. Figure 11(a) considers two instance sizes (as defined in Amazon EC2) and the price of renting one or four of them in AWS or in the CoC (one VM of similar size for each provider), together with the expected memory capacity (in number of 1KB metadata tuples) of such DepSpace setup. As can be seen in the figure, a setup with four Large instances would cost less than $1200 in the CoC per month while a similar setup in EC2 would cost $749. This difference of $451 can be seen as the operation cost of tolerating provider failures in our SCFS setup, and comes mainly from the fact that Rackspace and Elastichosts charge almost 100% more than EC2 and Azure for similar VM instances. Moreover, such costs can be factored among the users of the system, e.g., for one dollar per month, 2300 users can have a SCFS-CoC setup with Extra Large replicas for the coordination service. Finally, it is worth to mention that this fixed cost can be eliminated if the organization using SCFS hosts the coordination service in its own infrastructure.

Besides the fixed operation costs, each SCFS user has to pay for its usage (*executed operations* and *storage space*) of the file system. Figure 11(b) presents the cost of reading a file (open for read, read whole file and close) and writing a file (open for write, write the whole file, close) in SCFS-CoC and SCFS-AWS (S3FS and S3QL will have similar costs). The cost of reading a file is the only one that depends on the size of data, since providers charge around $0.12 per GB of outbound traffic, while inbound traffic is free. Besides that, there is also the cost of the `getMetadata` operation, used for cache validation, which is 11.32 microdollars ($\mu$$). This corresponds to the total cost of reading a cached file. The cost of writing is composed by metadata and lock service operations (see Figure 4), since inbound traffic is free. Notice that the design of SCFS exploits these two points: unmodified data is read locally and always written to the cloud for maximum durability.

VM Instance	EC2	EC2×4	CoC	Capacity
Large	$6.24	$24.96	$39.60	7M files
Extra Large	$12.96	$51.84	$77.04	15M files

(a) Operation costs/day and expected coordination service capacity.



(b) Cost per operation (log scale). (c) Cost per file per day.

Figure 11: The (fixed) operation and (variable) usage costs of SCFS. The costs include outbound traffic generated by the coordination service protocol for metadata tuples of 1KB.

Storage costs in SCFS are charged per number of files and versions stored in the system. Figure 11(c) shows the cost/version/day in SCFS-AWS and SCFS-CoC (considering the use of erasure codes and preferred quorums [15]). The storage costs of SCFS-CoC are roughly 50% more than of SCFS-AWS: two clouds store half of the file each while a third receives an extra block generated with the erasure code (the fourth cloud is not used).

It is also worth to mention that the cost of running the garbage collector corresponds to the cost of a list operation in each cloud ($\leq \mu$\$1/cloud), independently of the number of deleted files/versions. This happens because all used clouds do not charge delete operations.

5 Related Work

In this section we discuss some distributed file systems and cloud storage works that are most relevant to SCFS.

Cloud-backed file systems. S3FS [6] and S3QL [7] are two examples of cloud-backed file systems. Both these systems use unmodified cloud storage services (e.g., Amazon S3) as their backend storage. S3FS employs a blocking strategy in which every update on a file only returns when the file is written to the cloud, while S3QL writes the data locally and later pushes it to the cloud. An interesting design is implemented by BlueSky [39], another cloud-backed file system that can use cloud storage services as a storage backend. BlueSky provides a CIFS/NFS proxy (just as several commercially available cloud storage gateways) to aggregate writings in log segments that are pushed to the cloud in background, implementing thus a kind of log-structured cloud-backed file system. These systems differ from SCFS in many ways (see Figure 1), but mostly regarding their lack of controlled sharing support for geographically dispersed clients and dependency of a single cloud provider.

Some commercial cloud-enabled storage gateways [4, 5] also supports data sharing among proxies. These systems replicate file system metadata among the proxies, enabling one proxy to access files created by other proxies. Complex distributed locking protocols (executed by the proxies) are used to avoid write-write conflicts. In SCFS, a coordination service is used for metadata storage and lock management. Moreover, these systems neither support strongly consistent data sharing nor are capable to use a cloud-of-clouds backend.

Cloud-of-clouds storage. The use of multiple (unmodified) cloud storage services for data archival was first described in RACS [9]. The idea is to use RAID-like techniques to store encoded data in several providers to avoid vendor lock-in problems, something already done in the past, but requiring server code in the providers [31]. DepSky [15] integrates such techniques with secret sharing and Byzantine quorum protocols to implement single-writer registers tolerating arbitrary faults of storage providers. ICStore [12] showed it is also possible to build multi-writer registers with additional communication steps and tolerating only unavailability of providers. The main difference between these works and SCFS(-CoC) is the fact they provide a basic storage abstraction (a register), not a complete file system. Moreover, they provide strong consistency only if the underlying clouds provide it, while SCFS uses a consistency anchor (a coordination service) for providing strong consistency independently of the guarantees provided by the storage clouds.

Wide-area file systems. Starting with AFS [28], many file systems were designed for geographically dispersed locations. AFS introduced the idea of copying whole files from the servers to the local cache and making file updates visible only after the file is closed. SCFS adapts both these features for a cloud-backed scenario.

File systems like Oceanstore [32], Farsite [10] and WheelFS [36] use a small and fixed set of nodes as locking and metadata/index service (usually made consistent using Paxos-like protocols). Similarly, SCFS requires a small amount of computing nodes to run a coordination service and simple extensions would allow SCFS to use multiple coordination services, each one dealing with a subtree of the namespace (improving its scalability) [10]. Moreover, both Oceanstore [32] and Farsite [10] use PBFT [18] for implementing their metadata service, which makes SCFS-CoC superficially similar to their design: a limited number of nodes running a BFT state machine replication algorithm to support a metadata/coordination service and a large pool of untrusted storage nodes that archive data. However, on the contrary of these systems, SCFS requires few "explicit" servers, and only for coordination, since the storage nodes are replaced by cloud services like Amazon S3. Furthermore, these systems do not target controlled sharing of files and strong consistency, using thus long-term leases and weak cache coherence protocols. Finally, a distinctive feature of SCFS is that its design explicitly exploits the charging model of cloud providers.

6 Conclusions

SCFS is a cloud-backed file system that can be used for backup, disaster recovery and controlled file sharing, even without requiring trust on any single cloud provider. We built a prototype and evaluated it against other cloud-backed file systems and a file synchronization service, showing that, despite the costs of strong consistency, the design is practical and offers control of a set of tradeoffs related to security, consistency and cost-efficiency.

Acknowledgements. We thank the anonymous reviewers and Fernando Ramos for their comments to improve the paper. This work was supported by the EC's FP7 through projects BiobankCloud (317871) and TClouds (257243), and by the FCT through projects LaSIGE (PEst-OE/EEI/UI0408/2014) and INESC-ID (PEst-OE/EEI/LA0021/2013). Marcelo Pasin was funded by the EC's FP7 project LEADS (318809).

References

[1] 2012 future of cloud computing - 2nd annual survey results. `http://goo.gl/fyrZFD`.

[2] Amazon EC2 instance types. `http://aws.amazon.com/ec2/instance-types/`.

[3] Filebench webpage. `http://sourceforge.net/apps/mediawiki/filebench/`.

[4] Nasuni UniFS. `http://www.nasuni.com/`.

[5] Panzura CloudFS. `http://panzura.com/`.

[6] S3FS - FUSE-based file system backed by Amazon S3. `http://code.google.com/p/s3fs/`.

[7] S3QL - a full-featured file system for online data storage. `http://code.google.com/p/s3ql/`.

[8] S3QL 1.13.2 documentation: Known issues. `http://www.rath.org/s3ql-docs/issues.html`.

[9] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A case for cloud storage diversity. *SoCC*, 2010.

[10] A. Adya et al. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, 2002.

[11] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study o file-system metadata. In *FAST*, 2007.

[12] C. Basescu et al. Robust data sharing with key-value stores. In *DSN*, 2012.

[13] A. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga. DepSpace: A Byzantine fault-tolerant coordination service. In *EuroSys*, 2008.

[14] A. Bessani and M. Correia. Consistency anchor formalization and correctness proofs. Technical Report DI-FCUL-2014-02, ULisboa, May 2014.

[15] A. Bessani, M. Correia, B. Quaresma, F. Andre, and P. Sousa. DepSky: Dependable and secure storage in cloud-of-clouds. *ACM Trans. Storage*, 9(4), 2013.

[16] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia. On the efficiency of durable state machine replication. In *USENIX ATC*, 2013.

[17] A. Bessani, J. Sousa, and E. Alchieri. State machine replication for the masses with BFT-SMaRt. In *DSN*, 2014.

[18] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Trans. Computer Systems*, 20(4):398–461, 2002.

[19] S. Choney. Amazon Web Services outage takes down Netflix, other sites. `http://goo.gl/t9pRbX`, 2012.

[20] I. Drago et al. Inside Dropbox: Understanding personal cloud storage services. In *IMC*, 2012.

[21] G. Gibson et al. A cost-effective, high-bandwidth storage architecture. In *ASPLOS*, 1998.

[22] A. Grünbacher. POSIX access control lists on Linux. In *USENIX ATC*, 2003.

[23] J. Hamilton. On designing and deploying Internet-scale services. In *LISA*, 2007.

[24] J. Hamilton. Observations on errors, corrections, and trust of dependent systems. `http://goo.gl/LPTJoO`, 2012.

[25] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *SOSP*, 2011.

[26] M. Herlihy. Wait-free synchronization. *ACM Trans. Programing Languages and Systems*, 13(1):124–149, 1991.

[27] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programing Languages and Systems*, 12(3):463–492, 1990.

[28] J. Howard et al. Scale and performance in a distributed file system. *ACM Trans. Computer Systems*, 6(1):51–81, 1988.

[29] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale services. In *USENIX ATC*, 2010.

[30] F. Junqueira, B. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, 2011.

[31] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A durable and practical storage system. In *USENIX ATC*, 2007.

[32] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, 2000.

[33] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX ATC*, 2008.

[34] J. Li, M. N. Krohn, D. Mazieres, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.

[35] Z. Li et al. Efficient batched synchronization in dropbox-like cloud storage services. In *Middleware*, 2013.

[36] J. Stribling et al. Flexible, wide-area storage for distributed system with WheelFS. In *NSDI*, 2009.

[37] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking file system benchmarking: It *IS* rocket science. In *HotOS*, 2011.

[38] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[39] M. Vrable, S. Savage, and G. M. Voelker. BlueSky: A cloud-backed file system for the enterprise. In *FAST*, 2012.

[40] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.

[41] Y. Zhang, C. Dragga, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Viewbox: Integrating local file systems with cloud storage services. In *FAST*, 2014.