Growing Secure Distributed Systems From a Spore

Yunus Basagalar, Vassilios Lekakis, Pete Keleher
Department of Computer Science, University of Maryland
{yunusb,lex,keleher}@cs.umd.edu

***Abstract*—This paper describes the design and evaluation of Spore, a secure cloud-based file system that minimizes trust and functionality assumptions on underlying servers. Spore differs from other systems in that system relationships are formalized only through signed data objects, rather than in complicated protocols executed between clients and servers. This approach allows Spore to bootstrap a file system from a single object, providing integrity and security guarantees while storing all data as simple, immutable objects on untrusted servers.**

We use simulation to characterize the performance of this system, focusing primarily on the cost incurred in compensating for the minimal server support. We show that while a naive approach is quite inefficient, a series of simple optimizations can enable the system to perform well in real-world scenarios.

I. Introduction

Data "clouds" are useful for building highly-available, location-transparent access to data. Such clouds are usually based on large commercial server farms, or clusters of such farms. The resulting storage and bandwidth is therefore almost unlimited. Further, clouds are often both logically and geographically dispersed, preventing most single points of failure from threatening the entire system. For example, Google claims a record of five nines of uptime for its High Replication datastore as of May 2011 [1].

However, clouds are generally untrusted, are accessed over untrusted networks, and export differing guarantees and functionality. Cloud-based storage is inherently insecure, as the storage is under control of someone else (the cloud providers). While sophisticated and high-performing applications can and have been built on specific cloud systems using high-level services, the highest common denominator in the services of these systems is surprisingly low.

These two problems are intertwined. A great deal of recent research in distributed systems [2], [3], [4], [5] revolves around the idea of building secure, distributed object and file systems over remote data services, i.e., clouds. However, these systems usually rely on high-level guarantees and functionality specific to individual vendors. These can be used to export much of the protocol work, including the all-important task of ordering communication and updates, to the untrusted servers. These protocols structure the work in such a way that although important functionality occurs on untrusted hardware, mistrusting clients can verify the servers' performance after the fact.

The problem we are attacking is slightly different. Given the plethora of services, a natural question is to what extent the performance and trustworthiness of the entire system is dependent on these services. In other words, how low can service provider role be set without sacrificing performance and trust in the resulting system?

We provide a partial answer by describing and characterizing the performance of Spore, a secure distributed file system that attempts to minimize the trust and functionality it assumes in data stores. In particular, we assume every part of the system not under control of a single client is untrusted, including the network, servers, and other clients.

Spore avoids complex protocols and webs of trust between clients and servers by formalizing all system state, including read and write keys, server designations, and group membership, in a series of trust statements embedded in Spore objects. These same objects also include all file and directory data in the file system.

The system is bootstrapped from a single trusted object called the *spore*. The spore can designate a set of keys and objects as trusted, and these, in turn, can designate others as trusted. In effect, the file system is structured like an inductive proof, with trust in the entire system growing organically from the base case of a spore.

Spore assumes only a simple put/get/list interface (Section III-A) from the data sources it uses, a mapping between a name and an object. Further, Spore does not require objects to be mutable, and therefore may be implemented over write-once systems such as Chord [6]. This last assumption constrains the way objects are located, requiring the system to provide alternate methods of investing trust in objects that might not yet exist. However, it could also allow objects to be stored and disseminated in a variety of novel ways, include single-write media like CDs, embedded into an online image using steganographic techniques [7], or perhaps in the digital version of the New York Times.

Our main contributions, then, are two-fold:

- We describe the design of Spore, an existence proof of a trusted system built on untrusted distributed resources assumed to have only extremely limited functionality. Spore makes fewer assumptions about cloud functionality and trust than any comparable system of which we are aware.
- Second, we characterize the performance of a naive approach to Spore, which is quite poor. However, we show that this performance can be greatly improved through a series of simple optimizations: caching, periodic runs of *cleaner agents*, and probabilistic searches.

If Spore is able to provide trusted services, it will effectively provide a lower bound on the performance of systems with similar functionality in similar environments. As such systems take advantage of higher-level guarantees and functionality, their performance can only improve.

II. System Model

The purpose of the Spore project is to design a distributed file system that will allow a group of users to share data. The system is composed of client devices (which we here define as "a holder of a trusted public key") which access data on behalf of users, and a potentially malicious set of servers. The servers may be anything from cloud services, to local file systems, to peer-to-peer systems. They are not trusted, and not expected to export any higher-level functionality other than allowing immutable singleton objects to be stored and retrieved. The clients collaborate by storing objects on known servers, with names derived from agreed-upon conventions. In the aggregate, the clients construct a rooted *object graph* of the immutable objects. The graph embeds both the hierarchical structure of a file system, and a series of security statements that control access to objects lower (farther from the root) in the graph. Though not essential to the correctness of the system, the system may include *cleaner agents*, which are clients with trusted keys that perform maintenance operations on the object graph. A cleaner agent periodically traverses the graph, improving subsequent performance by creating new versions of directories that point to the most recent versions of their children. The cleaner agent could also be useful in building system-wide snapshots (Section III-F) and in supporting key revocation (Section IV-D).

A. Goals

We designed Spore to satisfy the following goals:

- **No assumptions about servers -** The system is designed to allow data to be stored on any type of server. We require only that servers allow objects to be stored, and later retrieved, using arbitrary names. Service-specific drivers could be used translate put and get requests into the API of individual servers. In particular, we do not require servers to implement arbitration or ordering services, as is expected in Sporc [4] or SUNDR [5], or connect objects in any way.
- **No assumptions about clients -** Clients interact with the rest of the system solely by reading and writing system objects. Clients do not communicate directly with each other except when communicating the location of the original spore, or when communicating symmetric keys for read access.
- **Minimize information leakage -** Eliminate information leakage to the extent possible. Connections and relationships between objects should not be divulged, for example. Though access to data cannot be disguised, the names used should not betray any information.
- **Data confidentiality -** Unauthorized users should not be able to access system data. Encrypting data protects against the server exposing data to unauthorized clients. However, we further prevent users whose access has been revoked from accessing future updates.
- **Tolerate forks -** Forks can occur whenever servers are not able to enforce strong notions of consistency. Our servers enforce nothing, forks must be tolerated, and mechanisms that can be used to resolve them should be provided.
- **Usable performance -** We are building this system on less capable building blocks previous system. The main goals have to do with flexibility, resilience, and security. Our performance goal is to show that a system can meet our objectives with performance good enough to be usable without requiring manual management.

Our goal is to use Spore to prevent malicious agents from preventing conforming clients from making progress. We assume Byzantine faults [8], [9]. In other words, malicious agents can be "insiders", i.e. holders of trusted keys, and they can cooperate.

All network and cloud resources are untrusted. For cloud resources, this means that any request can be either answered correctly, answered incorrectly, or ignored. Further, clouds are not trusted to protect the data.

We rely on the availability of strong cryptographic mechanisms, i.e., means of creating, using, and verifying public keys and signatures. We do not rely on a public key infrastructure (PKI).

A "client" is any entity able to find a spore and interpret its contents. In practice, a client will have at least a symmetric key for decrypting object data, and possibly a public key-pair to commit new writes to the system.

A "server" can be any service that returns set of objects (as an opaque bundle of bits) in response to a request with a specific name. Possible server types include cloud systems, peer-to-peer networks such as Chord [6], and local file systems.

III. Spore Design

This section describes the design of the Spore distributed file system. Though Spore is a file system, the principles discussed here are applicable to other types of distributed systems as well. We chose to implement a file system because the inherent need for versioning makes it an extremely challenging application to implement on untrusted storage, and using only immutable objects.

The system contains only a single type of entity: the *object*. Spore objects may be of any size. Since we assume only put/get/list interface (Section III-A), we do not rely on objects being mutable. This immutability simplifies reasoning about caching and replication.

Table I lists many of the types of data that may be materialized in a Spore object. These include file system meta-data, file contents, links to other objects, and the cryptographic keys and statements necessary to build a trusted system. Throughout this paper we discuss objects as if there are distinct types. However, a real implementation would have only a single type that can play many roles, sometimes many roles in the same object instance.

A. Versioning, and Naming

The immutability of objects necessarily implies that Spore is a *versioning* file system. Each modification to a "file" requires the creation of at least one new object containing the new changes, and requires some method of finding the new object versions from the old. Directories must also be

Name	Explanation
data	file system data
meta-data	file system meta-data
explicit links	to child objects, if a directory
implicit links	names of subsequent version objects
security statements	public keys, write keys, revocation, revocation lists, etc
hints	list of network addresses and protocols that might be useful in locating and accessing spore objects
snapshot	hashes precisely identifying reachable object versions
signature	valid signature of hash of the above

TABLE I: Data that can be included in an object. A spore is an object like any other, except that it is signed by the master key.

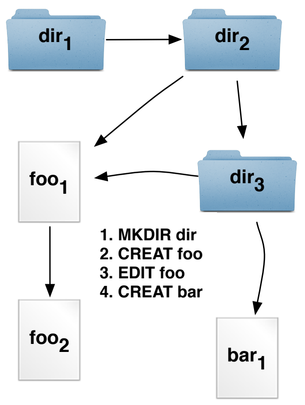


Fig. 1: Object graph resulting from (1) creation of $\mathtt{dir}_1$, (2) creating $\mathtt{foo}_1$ while updating `dir` to $\mathtt{dir}_2$, (3) editing `foo` to create $\mathtt{foo}_2$, and (4) creating $\mathtt{bar}_1$, and updating `dir` again. Note that the latest version of `dir` ($\mathtt{dir}_3$) still points to $\mathtt{foo}_1$.

versioned, new versions being created at least at each child creation or deletion. Conventional versioning file systems [10], [11], [12] avoid this issue by implementing directories as logs. However, a dynamic log cannot be implemented using a single immutable object.

Figure 1 describes a sequence of actions that create a directory containing two files, and the resulting *object graph* for this sequence. The file "foo" is created as version $\mathtt{foo}_1$; modifying this file creates a new object representing $\mathtt{foo}_2$. Each creation or deletion of a file also requires the directory to be versioned. Creating `foo` causes the creation of directory version $\mathtt{dir}_2$; creating `bar` causes $\mathtt{dir}_3$.

Assuming the sequence of actions described above, $\mathtt{foo}_2$ already existed before the last version of the directory was created. However, $\mathtt{foo}_2$ was not necessarily known to the client that creates $\mathtt{bar}_1$, so $\mathtt{dir}_3$ does not include a direct link to this version. Nonetheless, the latest version of `foo` can be found by traversing the graph regardless of the path taken.

This example shows directories, files, and versions tied together by directed edges. However, the parent of an object in this graph is necessarily created before the object itself. Given that the parent is immutable, the parent must somehow specify this link before its target exists.

We term this an *implicit edge*. Each object instantiating a specific file version specifies how to derive the name of the "next" version for that same file. The "latest" version of a file is found by traversing these edges until no valid object with the correct pre-defined name can be found.

Note that we have assumed so far that versions of a single logical file or directory are named sequentially, e.g., $\mathtt{foo}_1$ is followed by $\mathtt{foo}_2$. In practice, a secure wide-area file system might not wish to divulge the relationships between objects. Names are public because servers are untrusted, and can be expected to give out information without user authentication. We can instead use any arbitrary randomizing function to map from the name of one version to the next. The sole requirements are that this mapping is unpredictable and deterministic. In our proposed implementation, Spore objects name consecutive versions using an HMAC function with a supplied key. Creation of an object representing a new version of an existing file implies that the link from the file's parent directory object is now out of date. However, the latest version of a file can always be found by traversing implicit edges.

Spore tolerates forks by merging forked object versions with application-specific actions (Section III-C). However, the naming scheme discussed so far could allow concurrent updates to overwrite each other, and hence prevent applications from seeing all forked versions. Spore prevents this by appending the hash of an object's contents to the object's name as it is stored. With this scheme, an object corresponds to all objects in the storage service having the true object name as a prefix. As a result, retrieving an object requires a put/get/list interface with the server.

B. Skip lists

Clients find the "latest" version of a file by sequentially asking for each version, starting from the last version they might have cached. For example, given $\mathtt{foo}_2$ in the cache, a client will send requests for $\mathtt{foo}_3$, $\mathtt{foo}_4$, etc. until a version is not found. Runs of modifications to a single object can create a long series of version objects for a given file.

Spore probabilistically skips ahead in the search, with power-of-two jumps somewhat analogous to the way skip lists work. In the above example, the client might probabilistically check for $\mathtt{foo}_{18}$, then $\mathtt{foo}_{34}$, etc. This approach costs little, but helps greatly in minimizing the long tail of cost distributions.

C. Consistency

Tightly coupled systems can support strong notions of consistency by relying on servers to enforce orderings and arbitrate access. Spore servers have no higher level functionality. The strongest remaining consistency that can be supported is *fork consistency* [5]. Intuitively, fork consistency splits data views whenever two updates occur to the same object version. Fork consistency is the inevitable result of any system in which servers can misbehave.

Since Spore cannot prevent forks, it must tolerate them. Effectively, this means that the system must be able to tolerate reads returning multiple results. Forks can be automatically merged only when operations are known to be transitive, such as text edits (Sporc [4]) or distinct file creations and deletions (FICUS [13]).

We take the more general approach of allowing users with application-specific information to arbitrate among, or merge conflicting updates. Two conflicting updates can be merged by giving them a common successor. Recall that version $n+1$ of a file is, by default, named using a deterministic HMAC function of version n. By allowing this default to be overridden in special cases, conflicting updates can name the same successor.

Many distributed systems support eventual consistency [14] by either having a primary server for each object, or requiring servers to eventually agree on ordering. Spore has neither, and therefore cannot support eventual consistency.

Surprisingly, however, the causal relationships that comprise causal consistency [15] could be supported by encoding them in objects. For example, COPS [16] provides causal consistency by explicitly encoding dependencies among objects. Dependences are limited by differentiating among "contexts", which seem to correspond to distinct threads or processes. A Spore client could encode similar dependences automatically, allowing the system to support causal consistency.

D. Replication and Server Choice

Spore does not explicitly support replication, but immutable objects allow efficient client-directed replication because there is no need to maintain consistency across objects. Clients can therefore replicate objects by storing copies on multiple servers. Note that many servers, such as RAID-based local systems or remote clouds, are replicated internally. Since these servers are untrusted, however, the client should explicitly replicate high-value data.

The Spore object graph is location independent. If $\texttt{foo}_2$ is the name of the "next" version of `foo`, and properly signed object with that name will suffice. The object's location is not relevant.

On the other hand, location is relevant to a client trying to find the latest version of an object. We assume that objects encode sets of potential locations for successor objects. These sets should be small, as all must usually be checked when looking for the latest version of an object. However, searches at multiple sites can be done in parallel, and client-directed replication as above makes finding objects more likely.

E. The Cleaner

A spore system will often include at least one *cleaner agent*, which performs periodic maintenance on the object graph. An agent periodically traverses the graph, improving subsequent performance by creating new versions of directories that point to the most recent versions of their children. The cleaner agent can be any client that possesses a trusted key giving write access to the whole tree.

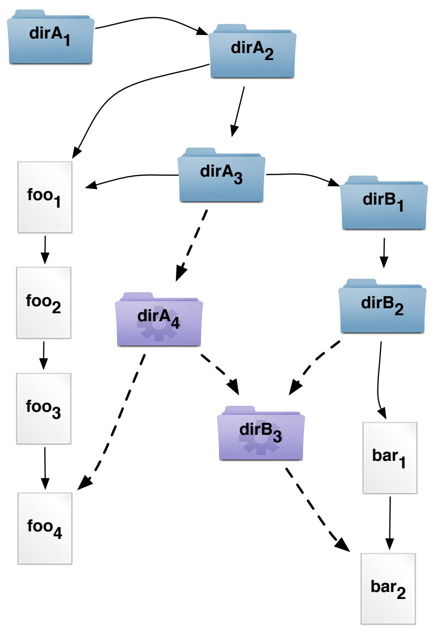


Fig. 2: The cleaner creates $\texttt{dirA}_4$ and $\texttt{dirB}_3$ to link the most recent directory versions to the most recent versions of contained files.

The example of Figure 2, minus $\texttt{dirA}_4$, $\texttt{dirB}_3$, and all dotted edges, could result from a simple sequence of write operations. Without $\texttt{dirA}_4$, each client attempting to retrieve the most recent version of `foo` must traverse each of the four versions, plus some number of directory versions. Similarly, retrieving `bar` without $\texttt{dirB}_3$ requires traversing all `dirA` objects, both `dirB` objects, and both `bar` versions. By adding new versions of `dirA` and `dirB`, the cleaner is able to reduce the path from the root to the most recent versions of both `foo` and `bar` slightly. More usefully, caching $\texttt{dirA}_4$ would make subsequent searches for `foo` and `bar` much more efficient.

The cleaner agent could also be useful in supporting snapshots (Section III-F), and key revocation (Section IV-D).

F. Immutability, and Snapshots

Spore detects tampering with objects by appending hashes of each object to its name. No client, authorized or not, can modify an object without the tampering being detected. However, objects can still be deleted by other clients on some servers, or deleted arbitrarily by servers, whether malicious or not.

While Spore cannot prevent deletions, it can probabilistically detect them. The mechanism is to gather lists (or graphs) of object names, together with hashes of their entire contents. Encoding such a list into a new object could be used to provide a snapshot either of the sub-graph rooted at that node, or the node's ancestors in the hierarchical file system space. For the latter, naming a version of a parent node effectively creates a snapshot of the parent directory at a specific time.

A complete run of the cleaner agent discussed above is a depth-first traversal of the entire graph, creating a new version of each directory on the way back up. When creating this directory, the cleaner could add hashes of children to the new versions of each directory. After a complete traversal, the aggregate of the per-directory hashes is equivalent to a Merkle tree with additional naming information. This is not quite a snapshot, as the data is still vulnerable to misbehaving servers. However, it encompasses sufficient information to precisely recreate the snapshot later *if* the data is still available.

IV. Security

Security in a distributed data system usually implies ways to prevent unauthenticated clients from reading or writing data. Spore differs in that servers, the holders of data, are neither trusted nor expected to implement any sophisticated functionality, even arbitration or ordering [5], [4]. Hence, servers do not necessarily perform client authentication before returning data, so the presumption must be that malicious outsiders can gain access to any object. We prevent breaches of data confidentiality through the usual expedient of encrypting all data with a symmetric cipher, such as AES. All objects need not be encrypted with the same key, but the keys used must be described further up the Spore graph.

Similarly, outsiders can also *create* new objects, even correctly encrypting them if they have read access to the system. However, these objects are only recognized as valid if they are signed with trusted keys, as described below. Hence, while outsiders can create new objects, these objects will not be recognized as part of the system.

A. Trust

The live system consists of a set of valid, trusted objects. The system is bootstrapped by initially investing trust in a single object: the *spore*. Each trusted object potentially contains data, as well as explicit or implicit descriptions of other trusted objects.

The security architecture is primarily a combination of principals (public keys) and statements (objects). Statements come in two flavors: *trust statements* and *structural statements*. Trust statements can specify that other keys and objects are also to be trusted. A trust statement might state that "key X is trusted", or that object "foo.c,v2" is *valid* if signed by a trusted key. By contrast, structural statements codify rules on how clients identify and interpret trusted objects and keys. For example, a structural statement might list data repositories where valid objects might be found.

Only statements and keys established in the ancestors of a given object are relevant for finding and interpreting that object. A key made trusted in $\texttt{bar}_1$ of Figure 1 is only useful for creating new versions of bar_1. Such a key could not be used for creating new versions of either bar_1's parent or *foo*, for example.

The goal of all these statements is to *allow conforming clients to avoid being misled by misbehaving clients.* The system has no way of preventing outsiders, or even a client with read but not write access, from creating correctly named objects. However, the lack of a trusted signature on such objects prevents clients from treating them as valid.

Read access is represented by possessing one or more symmetric keys. Write access is represented by holding the private key corresponding to a public key encoded in one of the system objects. The keys used for reading are either communicated out-of-band (any user's access to the system must be bootstrapped somehow), or encrypted with the user's public key and included in an object version.

In some sense, we are trying to construct a distributed system like an inductive proof. The base case consists of a *master key*, and the spore:

- *The master key is trusted* - Any object signed directly by the master key is a spore, creating a new file system.
- *A signed spore is trusted.* The spore sets up initial policies for the resulting file system: servers, public keys of initially trusted writers, the root directory.

Given the master key and the spore, the rest of the system grows organically using two types of inductive steps, as follows:

- *A trusted key can be used to assert that a given object is trusted* - An object is valid and trusted if and only if it is correctly signed by a trusted key *and* it conforms to rules established by ancestor objects. An example of the latter might be "any object with name "foo" is trusted if signed by a trusted key".
- *A trusted object can assert that a given key is trusted* - Objects are only trusted if signed by a trusted key. Lists of such keys can be defined in objects.

We distinguish between three types of keys in the system. The (single) master key is used to sign the initial spore, and is also used to make other keys trusted (able to create valid new objects), or untrusted (revocation). A *trusted* (write) key is one that allows a client to create and sign a new object in the system. Symmetric session keys are used to encrypt the data.

In more detail, trusted objects may make the following type of security-related statements:

- *public key validation* - As above, an object may state that a given public key is trusted and is usable for object creation, and/or making other keys trusted.
- *name validation* - An object, perhaps named "foo,v1", might state that any properly signed object named "foo,v2" is also part of the system, and trusted.
- *session keys* - An object might list a set of symmetric keys that can be used to encrypt and decrypt data.
- *revocation lists* - A revocation list contains a list of public keys not valid at objects reachable from the object containing the list. Only the master key can sign revocation lists.

B. Malicious Clients

Clients with write access must trust each other not to overwrite all objects with random data. We mitigate this need through a combination of a naming scheme and a versioning model (Section III-A). Even though misbehaving clients could inject erroneous data into the system, versioning would still provide earlier writes to and by conforming clients.

C. Malicious Servers

Servers can attempt to violate correctness by responding with incorrect data. Clients can avoid this problem by only accepting data with proper signatures.

A malicious server could also respond negatively to queries for data which it stores, or had stored previously. A client can detect the latter issue through probabilistic audits of data that it had previously stored [17], [18].

The most difficult to detect type of misbehavior is when a server intermittently claims not to have a specific object that, in fact, it does. Left unchecked, this type of misbehavior can result in forks of object version histories. The usual approach to detecting this misbehavior is to require clients and servers to sign all requests; two clients can later compare signed statements from the same server to see if they match [4]. However, this approach relies on servers being able to issue valid signatures and clients to communicate with each other, neither of which is an option in Spore. We purposefully require only low-level functionality from servers, and clients can identify each other only through public keys, not through Internet addresses.

Instead, we propose to use audits through anonymous third-party entities to verify server responses. The audits can take two forms. The first is similar to the first option discussed above. Clients can periodically audit servers to ensure data stored at the servers is being returned by queries. Requests would be sent through a third party in order to prevent the server from correlating the request with the previous store. The third party could be anything, including other clients (if known), Tor nodes, other local resources controlled by the same user, etc.

Clients cannot verify negative query responses by later attempting the same request through a third party, as the data could arrive in the interim. However, clients could use anonymous third parties to back-check positive responses to data that was stored by other clients. A subsequent negative response for the same data, through a third party, could indicate either that the server is lying, or that it has evicted the data. Either outcome is worth knowing.

These protocols would have to be cognizant of potential inconsistencies in the underlying server. For example, if one of the "servers" is a cloud service, a `put` and a subsequent `get` might be served by different nodes. A negative response to the `get` might lead us to erroneously conclude that the cloud provider is malicious. Azure does guarantee consistency among nodes [19], but others currently do not. However, services like S3 and Rackspace do appear to propagate updates throughout a region within a few seconds [20].

D. Key Revocation

Key revocation requires the establishment of a revocation list, which contains public keys that are no longer trusted. Inserting revocation lists high into the file system hierarchy would ensure that the revocations are seen by clients traversing the hierarchy, but Spore objects are immutable.

However, a single traversal of the cleaner agent potentially creates a new version for every directory in the file system hierarchy. In Figure 2, for example, a revocation list placed in $\texttt{dirA}_4$ could be in the path traversed by any client attempting to reach `foo` or `bar` from the root. One drawback of this approach is that it relies on constructing paths to leaf objects by preferring directory objects to file version objects. To reach the most recent version of `foo`, for example, the search should proceed from $\texttt{dirA}_2$ to $\texttt{dirA}_3$, rather than to $\texttt{foo}_1$. Revocation lists can be inserted into any of these new directory versions, including the latest version of the spore.

Revoked clients can be prevented from reading subsequent versions of an object by changing the session key used to encrypt content. The new session key can be encrypted with authorized public keys and included in a subsequent version.

Both skip lists and the cleaner agent can cause clients to miss key revocations. However, a client can probabilistically skip optimizations and traverse the whole hierarchy in order to avoid missing any security related updates. Additionally, we could create conventions about revocation list use, such as that the lists will be in a specific part of the namespace, or will be pushed as high as possible in the file system hierarchy. With this latter approach, clients would bias their probabilistic skips against skipping the high levels.

V. Performance Evaluation

In this section we evaluate the performance of a Spore file system via trace-driven simulation. The next sub-section describes our experimental setup, as well as the set of traces we used to drive the simulator. The rest of the section discusses individual experiments.

A. Setup

Simulation Platform: We ran all simulations on Linux machines with an Intel Core 2 Duo processor, 2GB RAM, and 500GB Western Digital 7200RPM hard drive.

Traces: Table II summarizes our three traces, each chosen to highlight different aspects of the system's performance.

The `emacs` trace consists of system calls made during a randomized copy of the emacs version 23 source tree to a new directory. The trace consists of approximately 4k system calls, dominated by `creat` and `write` calls.

The `rails` [21] trace is derived from developer activity on the Ruby on Rails web application framework. Rails source code is hosted on GitHub, a web-based hosting service based on the Git revision control system. According to the website [22], Rails is the most watched repository, as well as the third most forked, on their entire site. We extracted all changes to Rails over the last three years from the *git log*, and created a corresponding system call trace. Git logs contain only write operations. We use these writes as a backdrop to a synthetic reader trace based on the measured edit rate (popularity) of each file of the repository. From the popularities of the top 1000 files we constructed a synthetic trace averaging a single read for each two writes of the writers' trace. We chose

	Emacs	Rails		Coda
		Writers	Reader	
System calls	4510	64774	35106	47644
Dominant types	creat/write	write	read/access	read/write

TABLE II: Summary of the traces we used to drive the evaluation of Spore. The first row shows the number of systems calls that each trace consists. The second row shows the system call that dominates each trace

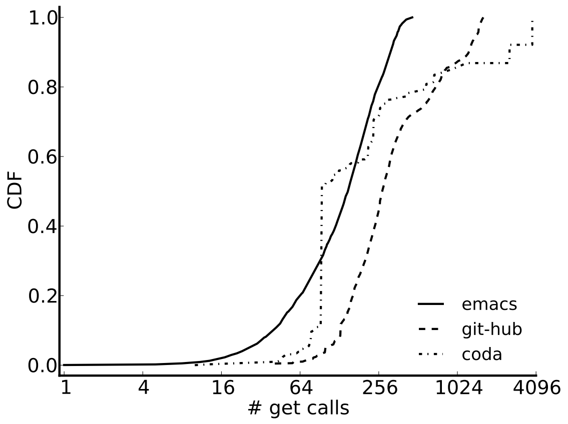


Fig. 3: Baseline: none of the simple optimizations are included.

this strategy because the git log does not contain entries for git-fetch/git-pull operations. The writers' trace contains 65k `write` system calls, while the reader trace contains 34k `read` and `access` system calls.

The `Coda` [23] trace is the write-intensive Dvorak trace from CMU [24]. This trace, while old, was chosen because it is derived from execution of a real distributed file system that targets disconnected operation and mobile computing. The Dvorak trace has the highest rate of file writes in of any of the available traces.

B. Simulator

Our simulator is implemented in Ruby 1.9.2 and it consists roughly of 2.5K lines of code. We implemented three optimizations in the simulator: client caches, the cleaner, and skip lists. Unfortunately, two of our traces omit file and write sizes, making simulation of a cache problematic. We therefore model the cache as being able to contain a specific number of whole files. Note that latencies in a wide-area system make local disks useful as a cache, effectively increasing cache size from several hundred megabytes to many gigabytes. However, none of our traces is able to benefit from caches this large.

We use three different cleaner load levels: `clean-none`, `clean-low`, and `clean-high`. `Clean-low` cleans a single directory at one twentieth the rate of a single client's file access rate, whereas `clean-high` means the cleaner cleans a directory at a rate of one to five.

We use a single metric to evaluate our approach: the number of "get" calls required to fetch a remote file. Clients in conventional systems retrieve an object by sending a single request to known server, thereby requiring only a single `get`, or remote message exchange. However, the simplifying assumptions of a Spore file system lead to potentially many versions of directories and files, each of which must be traversed at least once by the remote client. Figure 3 shows a CDF of the number of `gets` required to retrieve files, assuming no cache, no skip lists, and no cleaner. The majority of file fetches take more than one hundred `gets`, across all traces, and a few fetches take more than a thousand, clearly an unusably high number. Though the traces are deterministic, the skip list and cleaner techniques are probabilistic, so we average across five runs for each curve.

C. Backup

Figure 4a shows CDFs of various cache sizes for the `emacs` trace, without skip lists and with the cleaner off. Caching has an enormous effect on the number of `gets`, with medium cache allowing more than half of the fetches to take only a single `get`.

The figure also points out the additional pressure versioning puts on a Spore cache. Our simple cache model contains both file versions *and* directory versions, and does not distinguish between any of them in size (other than metadata). While each file in the copy has only a single version, each directory has as many versions as children.

Figures 4b and 5 show the effect of the cleaner and skip lists, respectively, with no cache. Cleaner activity creates new directory structures that point to the most recent versions of files. `Clean-low` results in fetches taking about half the time than without the cleaner, with a similar improvement to `clean-high`.

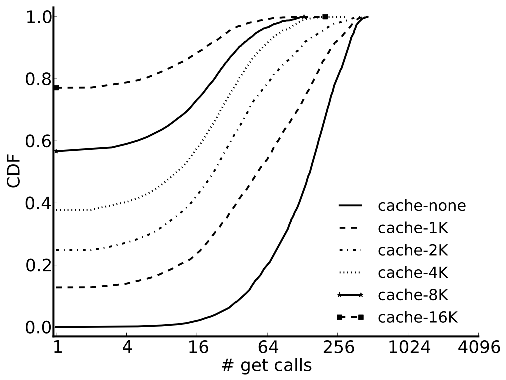
Skip lists make a slightly smaller improvement, an effect consistent across all of our traces. Skip lists do impose a cost (the extra probabilistic probes), but we never observed a situation where the aggregate cost outweighed the gain. We therefore use skip lists in the rest of our experiments.

D. Multiple Writers, Single Reader

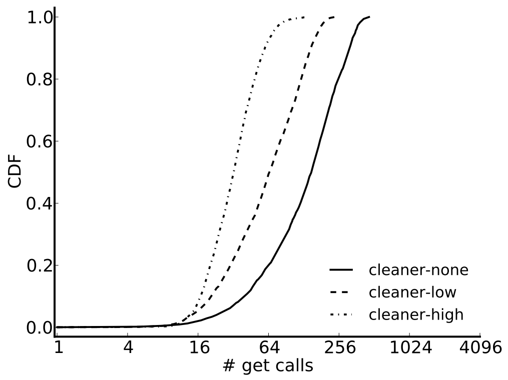
Figure 6 shows the performance impact of the cleaner and differing cache sizes on the `rails` GitHub trace. The high churn of the three-year traces makes larger cache sizes extremely useful. The cleaner has a very consistent effect of moving curves to the left, and helping to chop of the tail of distributions.

Several of the curves in Figure 6b have slight irregularities. In Figure 6b, for example, the 1k CDF briefly crosses the line of the 2k CDF, despite our use of a vanilla LRU cache replacement policy, which does not suffer from Belady's anomaly.

The explanation is that a file version in the cache affects the path chosen to reach the most recent version of a file. Consider the example in Figure 2 again, and assume that both

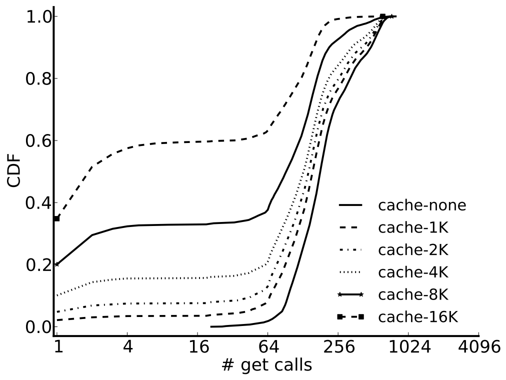


(a) No cleaner and no skip list.

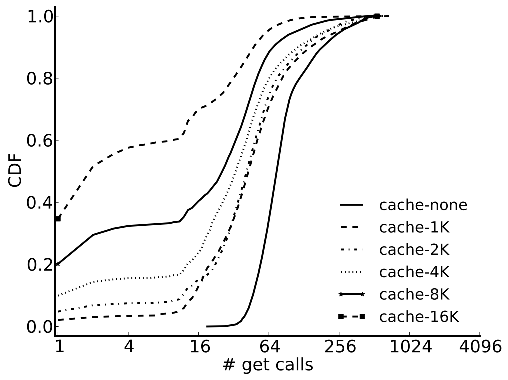


(b) No cache and no skip list.

Fig. 4: Absolute gains from different techniques in copying Emacs.



(a) With skip list but without cleaner.



(b) With skip list and cleaner.

Fig. 6: Sensitivity to cache size and the cleaner in the `rails` GitHub trace.

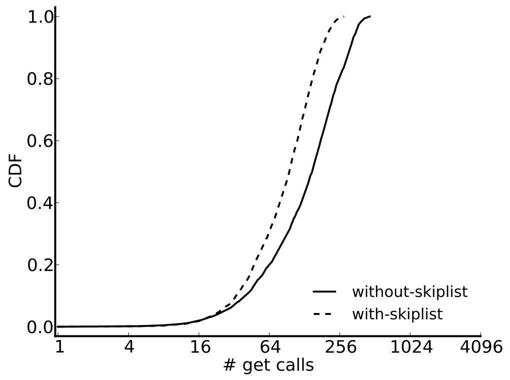


Fig. 5: Absolute gain for emacs with skip list.

$\texttt{foo}_1$ and $\texttt{dirA}_4$ are in the client's cache. Our current fetch procedure would proceed to $\texttt{foo}_2$, $\texttt{foo}_3$, and then to $\texttt{foo}_4$, when a better approach would be start from $\texttt{dirA}_4$.

Figure 7 shows cause for optimism that a Spore-like design would be usable in practice, as `coda` is a trace from a real distributed system. Even extremely small caches dramatically improve system performance. The cleaner reduces the number of `gets` on the low and no-cache cases, but an equally important effect is that of reducing the tails of the distributions by a factor of four or more, even with large caches.

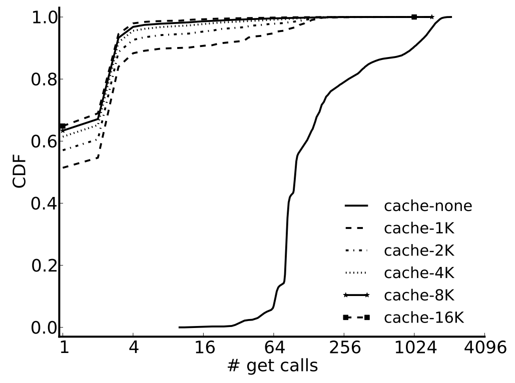
E. Discussion

Our primary metric is the number of `get` messages. We used this rather than latency to get a direct measurement of the improvements due to the different optimizations.

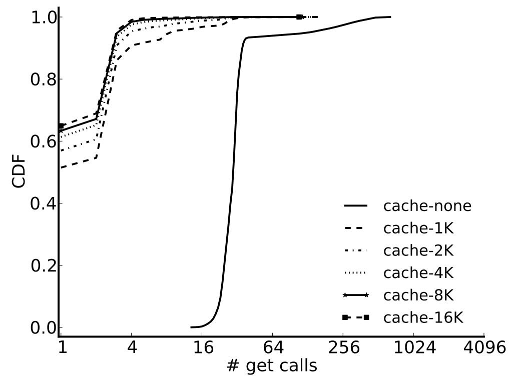
Spore's storage requirements are in direct proportion to object replication factors. This cost could be reduced by using erasure-coding, as in DepSky [25]. However, this would limit Spore's failure tolerance to a constant number.

The system performs poorly with a naive approach. However our results, especially with the `coda` trace, show that simple optimizations can dramatically improve that performance.

The skip lists have a relatively slight but always positive effect on performance. Analysis of our logs shows that this effect could be improved. Our current skip list approach wastes queries looking for versions very far in advance of the last



(a) With skip list but without cleaner.



(b) With skip list and cleaner.

Fig. 7: Relative performances of different cache sizes in Coda trace with respect to existence of cleaner.

known version. Adjusting probe distances to match measured runtime characteristics could eliminate most of this waste, while improving the hit rate.

The cleaner is also uniformly positive in effect, and would provide even more benefit to multiple clients. In addition to the performance gains, the cleaner also helps by creating snapshots, and in allowing revocation lists to be inserted high in the file hierarchy. Cleaners also create garbage, however, requiring more storage to hold all the cleaner-created data and adding to system cost. We theorize that some of this can be addressed through garbage collection.

Our approach includes a number of simplifications. First, the assumption of all files and directories taking approximately the same space in the cache is clearly unrealistic. The aggregate effect is probably to underestimate average performance, because the current approach allows directory versions (typically very small) to swamp the cache.

We currently do not cache file system data in the sense that is usually done. File attributes and data are usually cached in kernel buffers for some period of time, allowing repeated access to file meta-data to be satisfied with querying the file system proper. The simple strategy in this paper models a client that *always* checks for new versions, even if that exact check was done on the previous operation. Again, this leads us to underestimate system performance.

Our emphasis on performance from the point of view of a single client obscures the fact that, unlike caches and skip lists, the cleaner benefits all clients using the system. Our model also underestimates the effectiveness of the cleaner, as we currently model the cleaner as having the same cache size as clients. Though the cleaner's access would have a great deal of spatial locality, larger caches still improve the cleaner's effectiveness. Given its privileged status in the system, the cleaner's cache sizes should probably be decoupled from those of the clients'.

On the other hand, we do not model fetch attempts at multiple servers, and the cost of updating multiple servers. In both cases the queries can be parallelized, but would still result in increased overhead.

We also do not look at the performance costs of misbehaving servers.

VI. Related Work

The majority of our discussion of related work is sprinkled throughout this paper. To summarize, however, the most related prior work falls into two categories: cloud storage, and file system versioning.

SUNDR [5] and FAUST [2] were among the first projects that investigated untrusted servers. SUNDR defined the notion of fork consistency, and showed that a system of cooperating clients can limit the damage done by malicious server to that of creating unintended fork. Both inform clients about potential forks without being able to recover from them.

SPORC [4] provides an embedded access control mechanism that protects clients' privacy, and also proved the system resilient to all faults other than unintended forks. Additionally, SPORC uses *operational transformations* to merge forks automatically for specific types of applications.

Depot [3] builds a system from untrusted clouds. The system prevents malicious servers from doing harm by tight coupling between servers and clients, and between clients. Depot imposes high CPU and messaging overheads on clients.

DepSky [25] provides a storage service across multiple cloud providers. Their emphasis is on providing higher failure resilience by explicitly replicating each data object across multiple clouds. Spore's object placement is done by clients making a coherent erasure-coding technique difficult. However, systematic replication across servers, as in DepSky, could easily be implemented by clients.

These systems comprise the state of the art in cloud storage with untrusted servers. Spore differs in assuming no trust between any of the entities of the system. Servers are untrusted, clients do not trust or communicate with other clients. Prior systems rely on servers performing complex and important tasks, checking for correctness after the fact. Spore servers are as thin as possible, and rely on nothing more than a put/get/list interface over REST or SOAP, as is provided by most of the available commercial storage services [26].

Many systems have exposed versioning in the file system [27], [28], [11], [10], and described ways to minimize the attendant costs. However, most do so in the context of a single machine. Spore's wide-area nature precludes most of the optimizations discussed in these papers.

VII. Conclusions

Spore began as an exercise in designing a system with the least possible trust and functionality invested in servers. Somewhat surprisingly, we found that we could build a usable system by formalizing state entirely in signed and encrypted objects, and reasoning about system properties by following paths through the object graph. We do not advocate this architecture for general use, but feel that the minimal requirements might make it appropriate for a number of environments where other systems would not suffice.

The system is secure in that only clients with keys can read system data, and only authorized clients can write data that will be read by correct clients. Individual objects may be lost, as the underlying servers are untrusted, but forming names with hashes of object content allows correct clients to detect and avoid data that has been tampered with.

We are currently extending this work in a number of directions. The cleaner-based snapshots could be used to make probabilistic guarantees of file system integrity. Packing objects into larger segments [29] could reduce `gets`, especially for single-client or producer-consumer systems. Our skip list implementation needs to be tuned to runtime characteristics. Our current cache replacement algorithm is oblivious to the type and use of any of the objects. We might, for example, prefer directories to files, or prefer directories higher in the tree to those lower.

VIII. Acknowledgements

We would like to thank Bobby Bhattacharjee for helpful comments and insights, especially during the initial brainstorming phase of this project.

References

[1] Google, "Google app engine blog," http://googleappengine.blogspot.com/2011/05/app-engine-150-release.html, May 2011.

[2] C. Cachin, I. Keidar, and A. Shraer, "Fail-aware untrusted storage," IBM Zurich, Tech. Rep., 2008.

[3] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: cloud storage with minimal trust," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.

[4] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "Sporc: group collaboration using untrusted cloud resources," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.

[5] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (sundr)," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, 2004.

[6] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *SIGCOMM Comput. Commun. Rev.*, August 2001.

[7] R. Anderson, R. Needham, and A. Shamir, "The steganographic file system," in *Information Hiding*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1998.

[8] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the third symposium on Operating systems design and implementation*, ser. OSDI '99, 1999.

[9] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07, 2007.

[10] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir, "Deciding when to forget in the elephant file system," in *Proceedings of the seventeenth ACM symposium on Operating systems principles*, 1999.

[11] K.-K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok, "A versatile and user-oriented versioning file system," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004.

[12] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger, "Metadata efficiency in a comprehensive versioning file system," in *In Proceedings of USENIX Conference on File and Storage Technologies*, 2002.

[13] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek, "Resolving file conflicts in the ficus file system," in *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, ser. USTC'94, 1994.

[14] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," *SIGOPS Oper. Syst. Rev.*, 1995.

[15] M. Ahamad, P. W. Hutto, and R. John, "Implementing and programming causal distributed shared memory," in *Proceedings of the 11th International Conference on Distributed Computing Systems*, May 1991.

[16] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with cops," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, 2011.

[17] M. A. Shah, M. Baker, J. C. Mogul, and R. Swaminathan, "Auditing to keep online storage services honest," in *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, 2007.

[18] A. Haeberlen, P. Kuznetsov, and P. Druschel, "PeerReview: Practical accountability for distributed systems," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Oct 2007.

[19] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. ul Haq, M. I. ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows azure storage: a highly available cloud storage service with strong consistency," in *SOSP*, T. Wobber and P. Druschel, Eds. ACM, 2011, pp. 143–157. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043571

[20] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective," in *CIDR*, 2011.

[21] D. H. Hansson, "Ruby on rails," http://rubyonrails.org.

[22] GitHub, "Popular repositories," https://github.com/popular/watched.

[23] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A highly available file system for a distributed workstation environment," *IEEE Trans. Computing*, 1990.

[24] CMU, "Coda project traces," http://www.coda.cs.cmu.edu/DFSTrace.

[25] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "Depsky: dependable and secure storage in a cloud-of-clouds," in *Proceedings of the sixth conference on Computer systems*, 2011.

[26] J. Musser, "10 online storage apis," http://blog.programmableweb.com/2007/11/26/10-online-storage-apis/.

[27] K.-K. Muniswamy-Reddy and D. A. Holland, "Causality-based versioning," in *Proccedings of the 7th conference on File and storage technologies*, 2009.

[28] B. Cornell, P. A. Dinda, and F. E. Bustamante, "Wayback: a user-level versioning file system for linux," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '04, 2004.

[29] M. Vrable, S. Savage, and G. M. Voelker, "Cumulus: Filesystem backup to the cloud," *Trans. Storage*, vol. 5, pp. 14:1–14:28, December 2009. [Online]. Available: http://doi.acm.org/10.1145/1629080.1629084