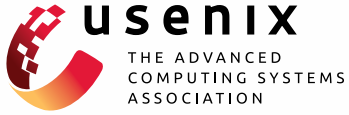

MetaSync: File Synchronization Across Multiple Untrusted Storage Services

Seungyeop Han and Haichen Shen, *University of Washington;* Taesoo Kim, *Georgia Institute of Technology;* Arvind Krishnamurthy, Thomas Anderson, and David Wetherall, *University of Washington*

https://www.usenix.org/conference/atc15/technical-session/presentation/han

This paper is included in the Proceedings of the
2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

Open access to the Proceedings of the
2015 USENIX Annual Technical Conference
(USENIX ATC '15) is sponsored by USENIX.

MetaSync: File Synchronization Across Multiple Untrusted Storage Services

Seungyeop Han, Haichen Shen, Taesoo Kim[†], Arvind Krishnamurthy, Thomas Anderson, and David Wetherall
University of Washington, [†]Georgia Institute of Technology

Abstract

Cloud-based file synchronization services, such as Dropbox, are a worldwide resource for many millions of users. However, individual services often have tight resource limits, suffer from temporary outages or even shutdowns, and sometimes silently corrupt or leak user data.

We design, implement, and evaluate MetaSync, a secure and reliable file synchronization service that uses multiple cloud synchronization services as untrusted storage providers. To make MetaSync work correctly, we devise a novel variant of Paxos that provides efficient and consistent updates on top of the unmodified APIs exported by existing services. Our system automatically redistributes files upon reconfiguration of providers.

Our evaluation shows that MetaSync provides low update latency and high update throughput while being more trustworthy and available. MetaSync outperforms its underlying cloud services by 1.2-10× on three realistic workloads.

1 Introduction

Cloud-based file synchronization services have become tremendously popular. Dropbox reached 300M users in May 2014, adding 100M customers in six months [15]. Many competing providers offer similar services, including Google Drive, Microsoft OneDrive, Box, and Baidu. These services provide very convenient tools for users, especially given the increasing diversity of user devices needing synchronization. With such resources and tools, mostly available for free, users are likely to upload ever larger amounts of personal and private data.

Unfortunately, not all services are trustworthy or reliable in terms of security and availability. Storage services routinely lose data due to internal faults [6] or bugs [13, 23, 30], leak users' personal data [12, 31], and alter user files by adding metadata [7]. They may block access to content (e.g., DMCA takedowns [38]). From time to time, entire cloud services may go out of business (e.g., Ubuntu One [9]).

Our work is based on the premise that users want file synchronization and the storage that existing cloud providers offer, but without the exposure to fragile, unreliable, or insecure services. In fact, there is no fundamental need for users to trust cloud providers, and given the above incidents our position is that users are best served by *not* trusting them. Clearly, a user may encrypt files before storing them in the cloud for confidentiality. More generally, Depot [27] and SUNDR [26] showed how to design systems from scratch in which users of the cloud storage obtain data confidentiality, integrity, and availability without trusting the underlying storage provider. However, these designs rely on fundamental changes to both client and server; our question was whether we could use existing services for these same ends?

Instead of starting from scratch, MetaSync provides file synchronization on top of multiple existing storage providers. We thus leverage resources that are mostly well-provisioned, normally reliable, and inexpensive. While each service provides unique features, their common purpose is to synchronize a set of files between personal devices and the cloud. By combining multiple providers, MetaSync provides users larger storage capacity, but more importantly a more highly available, trustworthy, and higher performance service.

The key challenge is to maintain a globally consistent view of the synchronized files across multiple clients, using only the service providers' unmodified APIs without any centralized server. We assume no direct client-client or server-server communication. To this end, we devise two novel methods: 1) pPaxos, an efficient client-based Paxos algorithm that maintains globally consistent state among multiple passive storage backends (§3.3), and 2) a stable deterministic replication algorithm that requires minimal reshuffling of replicated objects on service re-configuration, such as increasing capacity or even adding/removing a service (§3.4).

Putting it all together, MetaSync can serve users better in all aspects as a file synchronization service; users need trust only the software that runs on their own computers. Our prototype implementation of MetaSync, a ready-to-use open source project, currently works with five different file synchronization services, and it can be easily extended to work with other services.

2 Goals and Assumptions

The usage model of MetaSync matches that of existing file synchronization services such as Dropbox. A user configures MetaSync with account information for the underlying storage services, sets up one or more directories to be managed by the system, and shares each directory with zero or more other users. Users can connect these directories with multiple devices (we refer to the devices and software running on them as `clients` in this paper), and local updates are reflected to all connected

clients; conflicting updates are flagged for manual resolution. This usage model is supported by a background synchronization daemon (MetaSyncd in Figure 1).

For users desiring explicit control over the merge process, we also provide a manual git-like push/pull interface with a command line client. In this case, the user creates a set of updates and runs a script to apply the set. These sets of updates are atomic with respect to concurrent updates by other clients. The system accepts an update only if it has been merged with the latest version pushed by any client.

Our baseline design assumes the backend services to be curious, as well as potentially unreachable, and unreliable. The storage services may try to discover which files are stored along with their content. Some of the services may be unavailable due to network or system failures; some may accidentally corrupt or delete files. However, we assume that service failures are independent, services implement their own APIs correctly (except for losing and corrupting user data), and communications between client and server machines are protected. We also consider extensions to this baseline model where the services have faulty implementations of their APIs or are actively malicious (§3.6). Finally, we assume that clients sharing a specific directory are trusted, similar to a shared Dropbox directory today.

With this threat model, the goals of MetaSync are:

- **No direct client-client communication:** Clients coordinate through the synchronization services without any direct communication among clients. In particular, they never need to be online at the same time.
- **Availability:** User files are always available for both read and update despite any predefined number of service outages and even if a provider completely stops allowing any access to its previously stored data.
- **Confidentiality:** Neither user data nor the file hierarchy is revealed to any of the storage services. Users may opt out of confidentiality for better performance.
- **Integrity:** The system detects and corrects any corruption of file data by a cloud service, to a configurable level of resilience.
- **Capacity and Performance:** The system should benefit from the combined capacity of the underlying services, while providing faster synchronization and cloning than any individual service.

3 System Design

This section describes the design of MetaSync as illustrated by Figure 1. MetaSync is a distributed, synchronization system that provides a reliable, globally consistent storage abstraction to multiple clients, by using untrusted cloud storage services. The core library defines a generic cloud service API; all components are implemented on top of that abstraction. This makes it

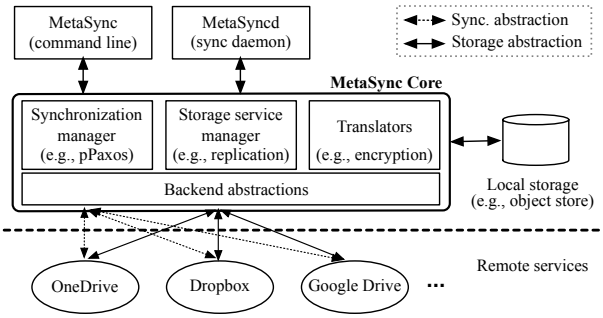

Figure 1: MetaSync has three main components: a storage service manager to coordinate replication; a synchronization manager to orchestrate cloud services; and translators to support data encryption. They are implemented on top of an abstract cloud storage API, which provides a uniform interface to storage backends. MetaSync supports two front-end interfaces: a command line interface and a synchronization daemon for automatic monitoring and check-in.

easy to incorporate a new storage service into our system (§3.7). MetaSync consists of three major components: synchronization manager, storage service manager, and translators. The synchronization manager ensures that every client has a consistent view of the user's synchronized files, by orchestrating storage services using pPaxos (§3.3). The storage service manager implements a deterministic, stable mapping scheme that enables the replication of file objects with minimal shared information, thus making our system resilient to reconfiguration of storage services (§3.4). The translators implement optional modules for encryption and decryption of file objects in services and for integrity checks of retrieved objects, and these modules can be transparently composed to enable flexible extensions (§3.5).

3.1 Data Management

MetaSync has a similar underlying data structure to that of git [20] in managing files and their versions: objects, units of data storage, are identified by the hash of their content to avoid redundancy. Directories form hash trees, similar to Merkle trees [29], where the root directory's hash is the root of the tree. This root hash uniquely defines a snapshot. MetaSync divides and stores each file into chunks, called Blob objects, in order to maintain and synchronize large files efficiently.

Object store. In MetaSync's object store, there are three kinds of objects—Dir, File and Blob—each uniquely identified by the hash of its content (with an object type as a prefix in Figure 2). A File object contains hash values and offsets of Blob objects. A Dir object contains hash values and names of File objects.

In addition to the object store, MetaSync maintains two kinds of metadata to provide a consistent view of the global state: *shared metadata*, which all clients can modify; and *per-client metadata*, which only the single owner (writer) client of the data can modify.

Shared metadata. MetaSync maintains a piece of shared metadata, called `master`, which is the hash value

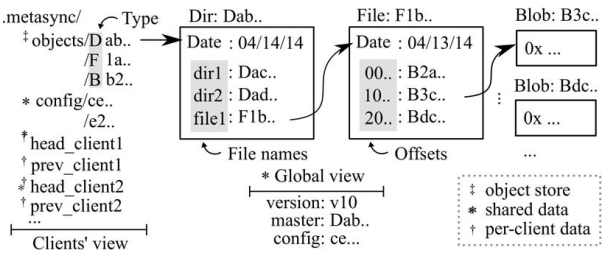

Figure 2: File management in a client's local directory. The object store maintains user files and directories with content-based addressing, in which the name of each object is based on the hash of its content. MetaSync keeps two kinds of metadata: shared, which all clients update; and per-client, for which the owner client is the only writer. The object store and per-client files can be updated without synchronization, while updates to the shared files require coordinated updates of the backend stores; this is done by the synchronization manager (§3.3).

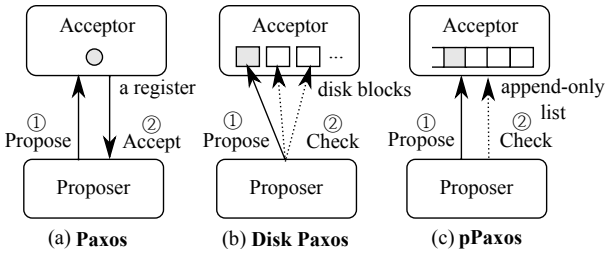

Figure 3: Comparison of operations between a proposer and an acceptor in Paxos [25], Disk Paxos [19], and pPaxos. Each acceptor in Paxos makes a local decision to accept or reject a proposal and then replies with the result. Disk Paxos assumes acceptors are passive; clients write proposals into per-client disk blocks at each acceptor. Proposers need to check every per-client block (at every acceptor) to determine if their proposal was accepted, or preempted by another concurrent proposal. With pPaxos, the append-only log allows clients to efficiently check the outcome at the passive acceptor.

of the root directory in the most advanced snapshot. It represents a consistent view of the global state; every client needs to synchronize its status against the `master`. Another shared piece of metadata is the configuration of the backend services including information regarding the list of backends, their capacities, and authenticators. When updating any of the shared metadata, we invoke a synchronization protocol built from the APIs provided by existing cloud storage providers (§3.3).

Per-client data. MetaSync keeps track of clients' states by maintaining a view of each client's status. The per-client metadata includes the last synchronized value, denoted as `prev_clientID`, and the current value representing the client's recent updates, denoted as `head_clientID`. If a client hasn't changed any files since the previous synchronization, the value of `prev_clientID` is equal to that of `head_clientID`. As this data is updated only by the corresponding client, it does not require any coordinated updates. Each client stores a copy of its per-client metadata into all backends after each update.

3.2 Overview

MetaSync's core library maintains the above data structures and exposes a reliable storage abstraction to applications. The role of the library is to mediate accesses and updates to actual files and metadata, and further interacts with the backend storage services to make file data persistent and reliable. The command line wrapper of the APIs works similarly with version control systems.

Initially, a user sets up a directory to be managed by MetaSync; files and directories under that directory will be synchronized. This is equivalent to creating a repository in typical version control systems. Then, MetaSync creates a metadata directory (`.metasync` as shown in Figure 2) and starts the synchronization of file data to backend services.

Each managed directory has a name (called namespace) in the system to be used in synchronizing with other clients. Upon initiation, MetaSync creates a folder with the name in each backend. The folder at the backend storage service stores the configuration information plus a subset of objects (§3.4). A user can have multiple directories with different configurations and composition of backends.

When files in the system are changed, an update happens as follows: (1) the client updates the local objects and `head_client` to point to the current root (§3.1); (2) stores the updated data blocks on the appropriate backend services (§3.4)); and (3) proposes its `head_client` value as the new value for `master` using pPaxos (§3.3)). The steps (1) and (2) do not require any coordination, as (1) happens locally and (2) proceeds asynchronously. Note that these steps are provided as separate functions to applications, thus each application or user can decide when to run each step; crucially, a client does not have to update global `master` for every local file write.

3.3 Consistent Update of Global View: pPaxos

The file structure described above allows MetaSync to minimize the use of synchronization operations. Each object in the object store can be independently uploaded as it uses content-based addressing. Each per-client data (e.g., `head_client_*`) is also independent since we ensure that only the owning client modifies the file. Thus, synchronization to avoid potential race conditions is necessary only when a client wants to modify shared data (i.e., `master` and configuration).

pPaxos. In a distributed environment, it is not straightforward to coordinate updates to data that can be modified by multiple clients simultaneously. To create a consistent view, clients must agree on the sequence of updates applied to the shared state, by agreeing on the next update applied at a given point.

Clients do not have communication channels between each other (e.g., they may be offline), so they need to rely on the storage services to achieve this consensus. However, these services do not communicate with each other,

nor do they implement consensus primitives. Instead, we devise a variant of Paxos [25], called pPaxos (passive Paxos) that uses the exposed APIs of these services.

We start our overview of pPaxos by relating it to the classic Paxos algorithm (see Figure 3(a)). There, each client works as a proposer and learner; the next state is determined when a majority accepts a given proposal. Acceptors act in concert to prevent inconsistent proposals from being accepted; failing proposals are retried. We cannot assume that the backend services will implement the Paxos acceptor algorithm. Instead, we only require them to provide an *append-only list* that *atomically* appends an incoming message at the end of the list. This abstraction is either readily available or can be layered on top of the interface provided by existing storage service providers (Table 3). With this append-only list abstraction, backend services can act as *passive acceptors*. Clients determine which proposal was "accepted" by examining the log of messages to determine what a normal Paxos acceptor would have done.

Algorithm. With an append-only list, pPaxos becomes a simple adaptation of classic Paxos, where the decision as to what proposal was accepted is performed by proposers. Each client keeps a data structure for each backend service, containing the state it would have if it processed its log as a Paxos acceptor (Figure 4 Lines 1-4). To propose a value, a client sends a `PREPARE` to every storage backend with a proposal number (Lines 7-8); this message is appended to the log at every backend. The proposal number must be unique (e.g., client IDs are used to break ties). The client determines the result of the prepare message by fetching and processing the logs at each backend (Lines 25-29). It aborts its proposal if another client inserted a larger proposal number in the log (Line 10). As in Paxos, the client proposes as the new root the value in the highest numbered proposal "accepted" by any backend server (Lines 12-15), or its own new root if none has been accepted. It sends this value in an `ACCEPT_REQ` message to every backend (Lines 18-19) to be appended to its log; the value is committed if no higher numbered `PREPARE` message intervenes in the log (Lines 20-21, 30-32). When the new root is accepted by a majority, the client can conclude it has committed the new updated value (Line 23). In case it fails, to avoid repeated conflicts the client chooses a random exponential back-off and tries again with an increased proposal number (Lines 33-36).

This setting is similar to the motivation behind Disk Paxos [19]; indeed, pPaxos can be considered as an optimized version of Disk Paxos (Figure 3(b)). Disk Paxos assumes that the storage device provides only a simple block interface. Clients write proposals to their own block on each server, but they must check everyone else's blocks to determine the outcome. Thus, Disk Paxos takes

```
1:  struct Acceptor
2:      round:      promised round number
3:      accepted:   all accepted proposals
4:      backend:    associated backend service
    [Proposer]
5:  procedure ProposeRound(value, round, acceptors)
     prepare:
6:     concurrently
7:     for all a ← acceptors do
8:         Send(⟨PREPARE, round⟩ → a.backend)
9:         Update(a)
10:        if a.round > round then abort
11:    wait until done by a majority of acceptors
     accept:
12:    accepted ← ∪_{a∈acceptors} a.accepted
13:    if |accepted| > 0 then
14:        p ← arg max{p.round | p ∈ accepted}
15:        value ← p.value
16:    proposal ← ⟨round, value⟩
17:    concurrently
18:    for all a ← acceptors do
19:        Send(⟨ACCEPT_REQ, proposal⟩ → a.backend)
20:        Update(a)
21:        if proposal ∉ a.accepted then abort
22:    wait until done by a majority of acceptors
     commit:
23:    return proposal
24: procedure Update(acceptor)
25:    log ← FetchNewLog(acceptor.backend)
26:    for all msg ∈ log do
27:        switch msg do
28:            case ⟨PREPARE, round⟩
29:                acceptor.round ← max(round, acceptor.round)
30:            case ⟨ACCEPT_REQ, proposal⟩
31:                if proposal.round ≥ acceptor.round then
32:                    acceptor.accepted.append(proposal)
33: procedure OnRestartAfterFailure(round)
34:    IncreaseRound
35:    WaitExponentailly
36:    ProposeRound(value, round, acceptors)
    [Passive Acceptor]
37: procedure OnNewMessage(⟨msg, round⟩)
38:    Append(⟨msg, round⟩ → log)
```

Figure 4: pPaxos Algorithm.

time proportional to the product of the number of servers and clients; pPaxos is proportional to number of servers.

pPaxos in action. MetaSync maintains two types of shared metadata: the `master` hash value and service configuration. Unlike a regular file, the configuration is replicated in all backends (in their object stores). Then, MetaSync can uniquely identify the shared data with a three tuple: `(version, master_hash, config_hash)`.

Version is a monotonically increasing number which is uniquely determined for each `master_hash`, `config_hash` pair. This tuple is used in pPaxos to describe the status of a client and is stored in `head_client` and `prev_client`.

The pPaxos algorithm explained above can determine and store the next value of the three tuple. Then, we build the functions listed in Table 1 by using a pPaxos instance per synchronized value. Each client keeps the last value

APIs	Description
propose(prev, next)	Propose a next value of prev. It returns the accepted next value, which could be next or some other value proposed by another client.
get_recent()	Retrieve the most recent value.

Table 1: Abstractions for consistent update.

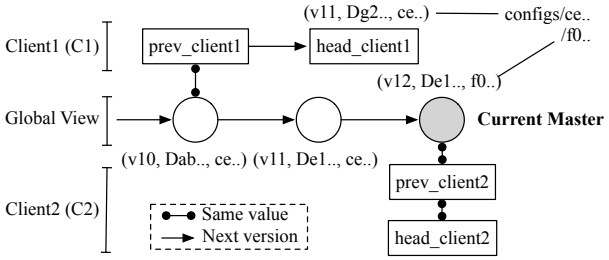

Figure 5: An example snapshot of pPaxos status with two clients. Each circle indicates a pPaxos instance. C1 synchronized against v10. It modified some files but the changes have not been synchronized yet (head_client1). C2 changed some files and the changes were made into v11, then made changes in configuration and synchronized it (v12). Then, it hasn't made any changes. If C1 tries to propose the next value of v10 later, it fails. It needs to merge with v12 and creates v13 head. In addition, C1 can learn configuration changes when getting v12.

with which it synchronized (`prev_client`). To proposes a new value, the client runs pPaxos to update the previous value with the new value. If another value has already been accepted, it can try to update the new value after merging with it. It can repeat this until it successfully updates the master value with its proposed one. This data structure can be logically viewed as a linked list, where each entry points the next hash value, and the tail of the list is the most up-to-date. Figure 5 illustrates an example snapshot of pPaxos status.

Merging. Merging is required when a client synchronizes its local changes (`head`) with the current master that is different from what the client previously synchronized (`prev`). In this case, proposing the current head as the next update to `prev` returns a different value than the proposed head as other clients have already advanced the `master` value. The client has to merge its changes with the current `master` into its head. To do this, MetaSync employs three-way merging as in other version control systems. This allows many conflicts to be automatically resolved. Of course, three-way merging cannot resolve all conflicts, as two clients may change the same parts of a file. In our current implementation, for example, MetaSync generates a new version of the file with `.conflict.N` extension, which allows for users to resolve the conflict manually.

3.4 Replication: Stable Deterministic Mapping

MetaSync replicates objects (in the object store) redundantly across R storage providers (R is configurable, typically $R = 2$) to provide high availability even when a service is temporarily inaccessible. This also provides potentially better performance over wide area networks. Since R is less than the number of services, it is required to maintain information regarding the mapping of objects to services. In our settings, where the storage services passively participate in the coordination protocol, it is particularly expensive to provide a consistent view of this shared information. Not only that, MetaSync requires a mapping scheme that takes into account storage space limits imposed by each storage service; if handled poorly, lack of storage at a single service can block the entire operation of MetaSync, and typical storage services vary in the (free) space they provide, ranging from 2 GB in Dropbox to 2 TB in Baidu. In addition, the mapping scheme should consider a potential reconfiguration of storage services (e.g., increasing storage capacity); upon changes, the re-balancing of distributed objects should be minimal.

Goals. Instead of maintaining the mapping information of each object, we use a stable, deterministic mapping function that locates each object to a group of services over which it is replicated; each client can calculate the same result independently given the same object. Given a hash of an object (mod H), the mapping is: map: $H \rightarrow \{s : |s| = R, s \subset S\}$, where H is the hash space, S is the set of services, and R is the number of replicas. The mapping should meet three requirements:

- R1 Support variations in storage size limits across different services and across different users.
- R2 Share minimal information amongst services.
- R3 Minimize realignment of objects upon removal or addition of a service.

To provide a balanced mapping that takes into account of storage variations of each service (R1), we may use a mapping scheme that represents storage capacity as the number of virtual nodes in a consistent hashing algorithm [24, 36]. Since it deterministically locates each object onto an identifier circle in the consistent hashing scheme, MetaSync can minimize information shared among storage providers (R2).

However, using consistent hashing in this way has two problems: an object can be mapped into a single service over multiple vnodes, which reduces availability even though the object is replicated, and a change in service's capacity—changing the number of virtual nodes, so the size of hash space—requires to reshuffle all the objects distributed across service providers (R3). To solve these problems, we introduce a stable, deterministic mapping scheme that maps an object to a unique set of virtual nodes and also minimizes reshuffling upon any changes to virtual nodes (e.g., changes in configurations). This construction is challenging because our scheme should randomly map each service to a virtual node and balance

```
1: procedure INIT(Services, H)
2:     ▷ H: HashSpace size, bigger values produce better mappings
3:     N ← {(sId, vId) : sId ∈ Services, 0 ≤ vId < Cap(sId)}
4:                         ▷ Cap: normalized capacity of the service
5:     for all i < H do map[i] = Sorted(N, key = md5(i, sId, vId))
6:     return map
7: procedure GETMAPPING(object, R)
8:     i ← hash(object) mod H
9:     return Uniq(map[i], R)    ▷ Uniq: the first R distinct services
```

Figure 6: The deterministic mapping algorithm.

object distribution, but at the same time, be stable enough to minimize remapping of replicated objects upon any change to the hashing space. The key idea is to achieve the random distribution via hashing, and achieve stability of remapping by sorting these hashed values; for example, an increase of storage capacity will change the order of existing hashed values by at most one.

Algorithm. Our stable deterministic mapping scheme is formally described in Figure 6. For each backend storage provider, it utilizes multiple virtual storage nodes, where the number of virtual nodes per provider is proportional to the storage capacity limit imposed by the provider for a given user. (The concept of virtual nodes is similar to that used in systems such as Dynamo [14].) Then it divides the hash space into H partitions. H is configurable, but remains fixed even as the service configuration changes. H can be arbitrary large but need to be larger than the sum of normalized capacity, with larger values producing better-balanced mappings for heterogeneous storage limits. During initialization, the mapping scheme associates differently ordered lists of virtual nodes with each of the H partitions. The ordering of the virtual nodes in the list associated with a partition is determined by hashing the index of the partition, the service ID, and the virtual node ID. Given an object hash n, the mapping returns the first R *distinct* services from the list associated with the (n mod H)th partition, similar to Rendezvous hashing [37].

The mapping function takes as input the set of storage providers, the capacity settings, value of H, and a hash function. Thus, it is necessary to share only these small pieces of information in order to reconstruct this mapping across different clients sharing a set of files. The list of services and the capacity limits are part of the service configuration and shared through the `config` file. The virtual node list is populated proportionally to service capacity, and the ordering in each list is determined by a uniform hash function. Thus, the resulting mapping of objects onto services should be proportional to service capacity limits with large H. Lastly, when N nodes are removed from or added to the service list, an object needs to be newly replicated into at most N nodes.

Example. Figure 7 shows an example of our mapping scheme with four services ($|S| = 4$) providing 1GB or 2GB of free spaces–for example, A(1) means that service A provides 1GB of free space. Given the replication requirement ($R = 2$) and the hash space ($H = 20$), we can populate the initial mapping with `Init` function from Figure 6. Subfigures (a) and (b) illustrate the realignment of objects upon the removal of service B(2) and the inclusion of a new service E(3).

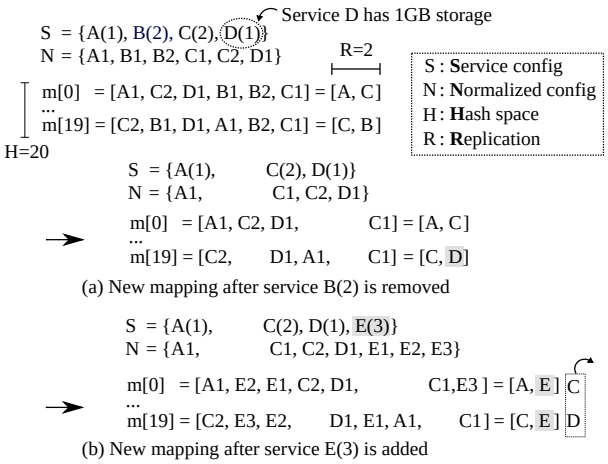

Figure 7: An example of deterministic mapping and its reconfigurations. The initial mapping is deterministically generated by Figure 6, given the configuration of four services, $A(1), B(2), C(2), D(1)$ where the number represents the capacity of each service. (a) and (b) show new mappings after configuration is changed. The grayed mappings indicate the new replication upon reconfiguration, and the dotted rectangle in (b) represents replications that will be garbage collected.

3.5 Translators

MetaSync provides a plugin system, called Translators, for encryption and integrity check. Translators is highly modular so can easily be extended to support a variety of other transformations such as compression. Plugins should implement two interfaces, `put` and `get`, which are invoked before storing objects to and after retrieving them from backend services. Plugins are chained, so that when an object is stored, MetaSync invokes a chain of `put` calls in sequence. Similarly, when an object is retrieved, it goes through the same chain but in reverse.

Encryption translator is currently implemented using a symmetric key encryption (AES-CBC). MetaSync keeps the encryption key locally, but does not store on the backends. When a user clones the directory in another device, the user needs to provide the encryption key. Integrity checker runs hash function over retrieved object and compares the digest against the file name. If it does not match, it drops the object and downloads the object by using other backends from the mapping. It needs to run only in the `get` chain.

3.6 Fault Tolerance

To operate on top of multiple storage services that are often unreliable (they are free!), faulty (they scan and tamper with your files), and insecure (some are outside of your country), MetaSync should tolerate faults.

Data model. By replicating each object into multiple backends (R in §3.4), MetaSync can tolerate loss of file or directory objects, and tolerate temporal unavailability or failures of $R-1$ concurrent services.

File integrity. Similarly with other version control systems [20], the hash tree ensures each object's hash value is valid from the root (`master, head`). Then, each object's integrity can be verified by calculating the hash of the content and comparing with the name when it is retrieved from the backend service. The value of `master` can be signed to protect against tampering. When MetaSync finds an altered object file, it can retrieve the data from another replicated service through the deterministic mapping.

Consistency control. MetaSync runs pPaxos for serializing updates to the shared value for `config` and `master`. The underlying pPaxos protocol requires $2f+1$ acceptors to ensure correctness if f acceptors may fail under the fail-stop model.

Byzantine Fault Tolerant pPaxos. pPaxos can be easily extended to make it resilient to other forms of service failures, e.g., faulty implementations of the storage service APIs and even actively malicious storage services. Note that even with Byzantine failures, each object is protected in the same way through replication and integrity checks. However, updates of global view need to be handled more carefully. We assume that clients are trusted and work correctly, but backend services may have Byzantine behavior. When sending messages for proposing values, a client needs to sign it. This ensures that malicious backends cannot create arbitrary log entries. Instead, the only possible malicious behavior is to break consistency by omitting log entries and reordering them when clients fetch them; a backend server may send any subset of the log entries in any order. Under this setting, pPaxos works similarly with the original algorithm, but it needs $3f+1$ acceptors when f may concurrently fail. Then, for each prepare or accept, a proposing client needs to wait until $2f+1$ acceptors have prepared or accepted, instead of $f+1$. It is easy to verify the correctness of this scheme. When a proposal gets $2f+1$ accepted replies, even if f of the acceptors are Byzantine, the remaining $f+1$ acceptors will not accept a competing proposal. As a consequence, competing proposals will receive at most $2f$ acceptances and will fail to commit. Note that each file object is still replicated at only $f+1$ replicas, as data corruption can be detected and corrected as long as there is a single non-Byzantine service. As a consequence, the only additional overhead of making the system tolerate Byzantine failures is to require a larger quorum ($2f+1$) and a larger number of storage services ($2f+1$) for implementing the synchronization operation associated with updating `master`.

APIs	Description
(a) Storage abstraction	
get(`path`)	Retrieve a file at `path`
put(`path`, `data`)	Store `data` at `path`
delete(`path`)	Delete a file at `path`
list(`path`)	List all files under `path` directory
poll(`path`)	Check if `path` was changed
(b) Synchronization abstraction	
append(`path`, `msg`)	Append `msg` to the list at `path`
fetch(`path`)	Fetch a log from `path`

Table 2: Abstractions for backend storage services.

3.7 Backend abstractions

Storage abstraction. Any storage service having an interface to allow clients to read and write files can be used as a storage backend of MetaSync. More specifically, it needs to provide the basis for the the functions listed in Table 2(a). Many storage services provide a developer toolkit to build a customized client accessing user files [16, 21]; we use these APIs to build MetaSync. Not only cloud services provide these APIs, it is also straightforward to build these functions on user's private servers through SSH or FTP. MetaSync currently supports backends with the following services: Dropbox, GoogleDrive, OneDrive, Box.net, Baidu, and local disk.

Synchronization abstraction. To build the primitive for synchronization, an append-only log, MetaSync can use any services that provide functions listed in Table 2(b). How to utilize the underlying APIs to build the append-only log varies across services. We summarize how MetaSync builds it for each provider in Table 3.

3.8 Other Issues

Sharing. MetaSync allows users to share a folder and work on the folder. While not many backend services have APIs for sharing functions—only Google Drive and Box have it among services that we used—others can be implemented through browser emulation. Once sharing invitation is sent and accepted, synchronization works the same way as in the one-user case. If files are encrypted, we assume that all collaborators share the encryption key.

Collapsing directory. All storage services manage individual files for uploading and downloading. As we see later in Table 4, throughput for uploading and downloading small files are much lower than those for larger files. As an optimization, we collapse all files in a directory into a single object when the total size is small enough.

4 Implementation

We have implemented a prototype of MetaSync in Python, and the total lines of code is about 7.5K. The current prototype supports five backend services including Box, Baidu, Dropbox, Google Drive and OneDrive,

and works on all major OSes including Linux, Mac and Windows. MetaSync provides two front-end interfaces for users, a command line interface similar to git and a synchronization daemon similar to Dropbox.

Abstractions. Storage services provide APIs equivalent to MetaSync's `get()` and `put()` operations defined in Table 2. Since each service varies in its support for the other operations, we summarize the implementation details of each service provider in Table 3. For implementing synchronization abstractions, `append()` and `fetch()`, we utilized the *commenting* features in Box, Google and OneDrive, and *versioning* features in Dropbox. If a service does not provide any efficient ways to support synchronization APIs, MetaSync falls back to the default implementation of those APIs that are built on top of their storage APIs, described for Baidu in Table 3. Note that for some services, there are multiple ways to implement the synchronization abstractions. In that case, we chose to use mechanisms with better performance.

Front-ends. The MetaSync daemon monitors file changes by using `inotify` in Linux, `FSEvents` and `kQueue` in Mac and `ReadDirectoryChangesW` in Windows, all abstracted by the Python library `watchdog`. Upon notification, it automatically uploads detected changes into backend services. It batches consecutive changes by waiting 3 more seconds after notification so that all modified files are checked in as a single commit to reduce synchronization overhead. It also polls to find changes uploaded from other clients; if so, it merges them into the local drive. The command line interface allows users to manually manage and synchronize files. The usage of MetaSync commands is similar to that of version control systems (e.g., `metasync init`, `clone`, `checkin`, `push`).

5 Evaluation

This section answers the following questions:

- What are the performance characteristics of pPaxos?
- How quickly does MetaSync reconfigure mappings as services are added or removed?
- What is the end-to-end performance of MetaSync?

Each evaluation is done on Linux servers connected to campus network except for synchronization performance in §5.3. Since most services do not have native clients for Linux, we compared synchronization time for native clients and MetaSync on Windows desktops.

Before evaluating MetaSync, we measured the performance variance of services in Table 4 via their APIs. One important observation is that all services are slow in handling small files. This provides MetaSync the opportunity to outperform them by combining small objects.

5.1 pPaxos performance

We measure how quickly pPaxos reaches consensus as we vary the number of concurrent proposers. The results of the experiment with 1-5 proposers over 5 storage providers are shown in Figure 8. A single run of pPaxos took about 3.2 sec on average under a single writer model to verify acceptance of the proposal when using all 5 storage providers. This requires at least four round trips: `PREPARE` (Send, FetchNewLog) and `ACCEPT_REQ` (Send, FetchNewLog) (Figure 4) (there could be multiple rounds in FetchNewLog depending on the implementation for each service). It took about 7.4 sec with 5 competing proposers. One important thing to emphasize is that, even with a slow connection to Baidu, pPaxos can quickly be completed with a single winner of that round. Also note that when compared to a single storage provider, the latency doesn't degrade with the increasing number of storage providers—it is slower than using a certain backend service (Google), but it is similar to the median case as the latency depends on the proposer getting responses from the majority.

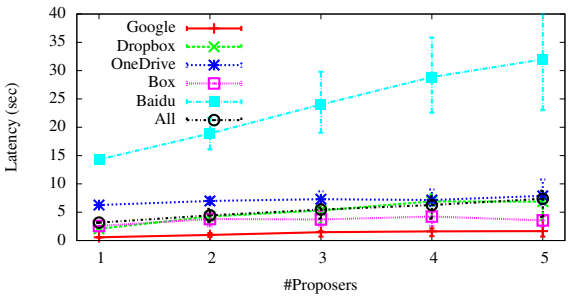

Figure 8: Latency (sec) to run a single pPaxos round with combinations of backend services and competing proposers: when using 5 different storage providers as backend nodes (all), the common path of pPaxos at a single proposer takes 3.2 sec, and the slow path with 5 competing proposers takes 7.4 sec in median. Each measurement is done 5 times, and it shows the average latency.

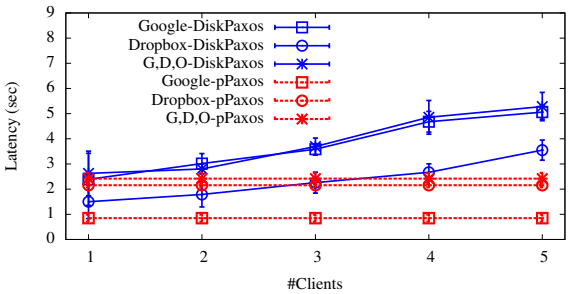

Figure 9: Comparison of latency (sec) to run a single round for Disk Paxos and pPaxos with varying number of clients when only one client proposes a value. Each line represents different backend setting; G,D,O: Google, Dropbox, and Onedrive. While pPaxos is not affected by the number of clients, Disk Paxos latency increases with it. Each measurement is done 5 times, and it shows the average latency.

Next, we compare the latency of a single round for pPaxos with that for Disk Paxos [19]. We build Disk Paxos with APIs by assigning a file as a block for each client. Figure 9 shows the results with varying number of clients when only one client proposes a value. As we explain in §3.3, Disk Paxos gets linearly slower with increasing number of clients even when all other clients are

Service	Synchronization API		Storage API
	`append(path, msg)`	`fetch(path)`	`poll(path)`
Box Google OneDrive	Create an empty `path` file and add `msg` as *comments* to the `path` file.	Download the entire comments attached on the `path` file.	Use `events` API, allowing long polling. (Google, OneDrive: periodically list `pPaxos` directory to see if any changes)
Baidu	Create a `path` directory, and consider each file as a log entry containing `msg`. For each entry, we create a file with an increasing sequence number as its name. If the number is already taken, we will get an exception and try with a next number.	List the `path` directory, and download new log entries since last fetch (all files with subsequent sequence numbers).	Use `diff` API to monitor if there is any change over the user's drive.
Dropbox	Create a `path` file, and overwrite the file with a new log entry containing `msg`, relying on Dropbox's versioning.	Request a list of versions of the `path` file.	Use `longpoll_delta`, a blocked call, that returns if there is a change under `path`.
Disk†	Create a `path` file, and append `msg` at the end of the file.	Read the new entries from the `path` file.	Emulate long polling with a condition variable.

Table 3: Implementation details of synchronization and storage APIs for each service. Note that implementations of other storage APIs (e.g., `put()`) can be directly built with APIs provided by services, with minor changes (e.g., supporting namespace). Disk† is implemented for testing.

Services	1 KB		1 MB		10 MB		100 MB	
	U.S.	China	U.S.	China	U.S.	China	U.S.	China
Baidu	0.7 / 0.8	1.8 / 2.6	0.21 / 0.22	0.12 / 1.48	0.22 / 0.94	0.13 / 2.64	0.24 / 1.07	0.13 / 3.38
Box	1.4 / 0.6	0.8 / 0.2	0.73 / 0.44	0.11 / 0.12	4.79 / 3.38	0.13 / 0.68	17.37 / 15.77	0.13 / 1.08
Dropbox	1.2 / 1.3	0.5 / 0.5	0.59 / 0.69	0.10 / 0.20	2.50 / 3.48	0.09 / 0.41	3.86 / 14.81	0.13 / 0.68
Google	1.4 / 0.8	-	1.00 / 0.77	-	5.80 / 5.50	-	9.43 / 26.90	-
OneDrive	0.8 / 0.5	0.3 / 0.1	0.45 / 0.34	0.01 / 0.05	3.13 / 2.08	0.11 / 0.12	7.89 / 6.33	0.11 / 0.44
	KB/s		MB/s		MB/s		MB/s	

Table 4: Upload and download bandwidths of four different file sizes on each service from U.S. and China. This preliminary experiment explains three design constrains of MetaSync. First, all services are extremely slow in handling small files, 7k/34k times slower in uploading/downloading 1 KB files than 100 MB on Google storage service. Second, the bandwidth of each service approaches its limit at 100 MB. Third, performance varies with locations, 30/22 times faster in uploading/downloading 100 MB when using Dropbox in U.S. compared to China.

inactive, since it must read the current state of all clients.

5.2 Deterministic mapping

We then evaluate how fairly our deterministic mapping distributes objects into storage services with different capacity, in three replication settings ($R = 1, 2$). We test our scheme by synchronizing source tree of Linux kernel 3.10.38, consisting of a large number of small files (464 MB), to five storage services, as detailed in Table 5. We use $H = (5 \times \text{sum of normalized space}) = 10,410$ for this testing. In $R = 1$, where we upload each object once, MetaSync locates objects in balance to all services—it uses 0.02% of each service's capacity consistently. However, since Baidu provides 2TB (98% of MetaSync's capacity in this configuration), most of the objects will be allocated into Baidu. This situation improves for $R = 2$, since objects will be placed into other services beyond Baidu. Baidu gets only 6.2 MB of more storage when increasing $R = 1 \rightarrow 2$, and our mapping scheme preserves the balance for the rest of services (using 1.3%).

The entire mapping plan is deterministically derived from the shared `config`. The size of information to be shared is small (less than 50B for the above example), and the size of the populated mapping is about 3MB.

Reconfiguration	#Objects Added / Removed	Time (sec) Replication / GC
$S = 4, R = 2 \rightarrow 3$	101 / 0	33.7 / 0.0
$S = 4 \rightarrow 3, R = 2$	54 / 54	19.6 / 40.6
$S = 3 \rightarrow 4, R = 2$	54 / 54	29.8 / 14.7

Table 6: Time to relocate 193 MB amount of objects (photo-sharing workloads in Table 7) on increasing the replication ratio, removing an existing service, and adding one more service. MetaSync quickly rebalances its mapping (and replication) based on its new `config`. We used four services, Dropbox, Box, GoogleDrive, and OneDrive ($S = 4$) for experimenting with the replication, including ($S = 3 \rightarrow 4$) and excluding OneDrive ($S = 4 \rightarrow 3$) for re-configuring storage services.

The relocation scheme is resilient to changes as well, meaning that redistribution of objects is minimal. As in Table 6, when we increased the configured replication by one ($R = 2 \rightarrow 3$) with 4 services, MetaSync replicated 193 MB of objects in about half a minute. When we removed a service from the configuration, MetaSync redistributed 96.5 MB of objects in about 20 sec. After adding and removing a storage backend, MetaSync needs to delete redundant objects from the previous configuration, which took 40.6/14.7 sec for removing/adding OneDrive in our experiment. However, the garbage collection will be asynchronously initiated during idle time.

Repl.	Dropbox (2 GB)	Google (15 GB)	Box (10 GB)	OneDrive (7 GB)	Baidu (2048 GB)	Total (2082 GB)
$R = 1$	77 (0.09%) 0.34 MB (0.02%)	660 (0.75%) 2.87 MB (0.02%)	475 (0.54%) 2.53 MB (0.02%)	179 (0.20%) 0.61 MB (0.01%)	86,739 (98.42%) 463.8 MB (0.02%)	88,130 (100%) 470.1 MB (0.02%)
$R = 2$	5,297 (3.01%) 27.4 MB (1.34%)	39,159 (22.22%) 206.4 MB (1.34%)	25,332 (14.37%) 138.2 MB (1.35%)	18,371 (10.42%) 98.3 MB (1.37%)	88,101 (49.98%) 470.0 MB (0.02%)	176,260 (100%) 940.3 MB (0.04%)

Table 5: Replication results by our deterministic mapping scheme (§3.4) for Linux kernel 3.10.38 (Table 7) on 5 different services with various storage space, given for free. We synchronized total 470 MB of files, consisting of 88k objects, and replicated them across all storage backends. Note that for this mapping test, we turned off the optimization of collapsing directories. Our deterministic mapping distributed objects in balance: for example, in $R = 2$, Dropbox, Google, Box and OneDrive used consistently 1.35% of their space, even with 2-15 GB of capacity variation. Also, $R = 1$ approaches to the perfect balance, using 0.02% of storage space in all services.

5.3 End-to-end performance

We selected three workloads to demonstrate performance characteristics. First, Linux kernel source tree (2.6.1) represents the most challenging workload for all storage services due to its large volume of files and directory (920 directories and 15k files, total 166 MB). Second, MetaSync's paper represents a causal use of synchronization service for users (3 directories and 70 files, total 1.6 MB). Third, sharing photos is for maximizing the throughput of each storage service with bigger files (50 files, total 193 MB).

Table 7 summarizes our results for end-to-end performance for all workloads, comparing MetaSync with the native clients provided by each service. Each workload was copied into one client's directory before synchronization is started. The synchronization time was measured as the length of interval between when one desktop starts to upload files and the creation time of the last file synced on the other desktop. We also measured the synchronization time for all workloads by using MetaSync with different settings. MetaSync outperforms any individual service for all workloads. Especially for Linux kernel source, it took only 12 minutes when using 4 services (excluding Baidu located outside of the country) compared to more than 2 hrs with native clients. This improvement is possible due to using concurrent connections to multiple backends, and optimizations like collapsing directories. Although these native clients may not be optimized for the highest possible throughput, considering that they may run as a background service, it would be beneficial for users to have a faster option. It is also worth noting that replication helps sync time, especially when there is a slower service, as shown in the case with $S = 5, R = 1, 2$; a downloading client can use faster services while an uploading client can upload a copy in the background.

Clone. Storage services often limit their download throughput: for example, MetaSync can download at 5.1 MB/s with Dropbox as a backend, and at 3.4 MB/s with Google Drive, shown in Figure 10. Note that downloading is done already by using concurrent connections even to the same service. By using multiple storage services, MetaSync can fully exploit the bandwidth of local connection of users, not limited by the allowed throughput of each service. For example, MetaSync with both services and R=2 took 25.5 sec for downloading 193 MB data, which is at 7.6 MB/s.

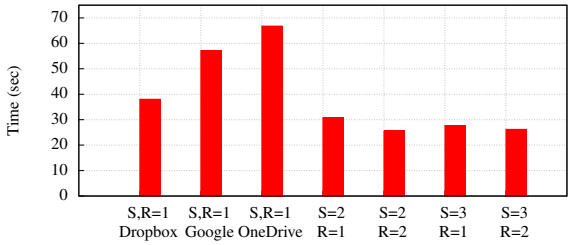

Figure 10: Time to clone 193 MB photos. When using individual services as a backend (Dropbox, Google, and OneDrive), MetaSync took 40-70 sec to clone, but improved the performance, 25-30 sec (30%) by leveraging the distributions of objects across multiple services.

6 Related Work

A major line of related work, starting with Farsite [2] and SUNDR [26] but carrying through SPORC [17], Frientegrity [18], and Depot [27], is how to provide tamper resistance and privacy on untrusted storage server nodes. These systems assume the ability to specify the client-server protocol, and therefore cannot run on unmodified cloud storage services. A further issue is equivocation; servers may tell some users that updates have been made, and not others. Several of these systems detect and resolve equivocations after the fact, resulting in a weaker consistency model than MetaSync's linearizable updates. A MetaSync user knows that when a push completes, that set of updates is visible to all other users and no conflicting updates will be later accepted. Like Farsite, we rely on a stronger assumption about storage system behavior—that failures across multiple storage providers are independent, and this allows us to provide a simpler and more familiar model to applications and users.

Likewise, several systems have explored composing a storage layer on top of existing storage systems. Syndicate [32] is designed as an API for applications; thus, they delegate design choices such as how to manage files and replicate to application policy. SCFS [5] imple-

Workload	Dropbox	Google	Box	OneDrive	Baidu	MetaSync			
						$S=5,R=1$	$S=5,R=2$	$S=4,R=1$	$S=4,R=2$
Linux kernel source	2h 45m	> 3hrs	> 3hrs	2h 03m	> 3hrs	1h 8m	13m 51s	18m 57s	12m 18s
MetaSync paper	48	42	148	54	143	55	50	27	26
Photo sharing	415	143	536	1131	1837	1185	180	137	112

Table 7: Synchronization performance of 5 native clients provided by each storage service, and with four different settings of MetaSync. For $S=5,R=1$, using all of 5 services without replication, MetaSync provides comparable performance to native clients–median speed for MetaSync paper and photo sharing, but outperforming for Linux kernel workloads. However, for $S=5,R=2$ where replicating objects twice, MetaSync outperform >10 times faster than Dropbox in Linux kernel and 2.3 times faster in photo sharing; we can finish the synchronization right after uploading a single replication set (but complete copy) and the rest will be scheduled in background. To understand how slow straggler (Baidu) affects the performance ($R=1$), we also measured synchronization time on $S=4$ without Baidu, where MetaSync vastly outperforms all services.

ments a sharable cloud-backed file system with multiple cloud storage services. Unlike MetaSync, Syndicate and SCFS assume separate services for maintaining metadata and consistency. RACS [1] uses RAID-like redundant striping with erasure coding across multiple cloud storage providers. Erasure coding can also be applied to MetaSync and is part of our future work. SpanStore [39] optimizes storage and computation placement across a set of paid data centers with differing charging models and differing application performance. As they are targeting general-purpose infrastructure like EC2, they assume the ability to run code on the server. BoxLeech [22] argues that aggregating cloud services might abuse them especially given a user may create many free accounts even from one provider, and demonstrates it with a file sharing application. GitTorrent [3] implements a decentralized GitHub hosted on BitTorrent. It uses BitCoin's blockchain as a method of distributed consensus.

Perhaps closest to our intent is DepSky [4]; it proposes a cloud of clouds for secure, byzantine-resilient storage, and it does not require code execution on the servers. However, they assume a more restricted use case. Their basic algorithm assumes at most one concurrent writer. When writers are at the same local network, concurrent writes are coordinated by an external synchronization service like ZooKeeper. Otherwise, it has a possible extension that can support multiple concurrent updates without an external service, but it requires clock synchronization between clients. MetaSync makes no clock assumptions about clients, it is designed to be efficient in the common case where multiple clients are making simultaneous updates, and it is non-blocking in the presence of either client or server failures. DepSky also only provides strong consistency for individual data objects, while MetaSync provides strong consistency across all files in a repository.

Our implementation integrates and builds on the ideas in many earlier systems. Obviously, we are indebted to earlier work on Paxos [25] and Disk Paxos [19]; we earlier provided a detailed evaluation of these different approaches. We maintain file objects in a manner similar to a distributed version control system like git [20]; the Ori file system [28] takes a similar approach. However, MetaSync can combine or split each file object for more efficient storage and retrieval. Content-based addressing has been used in many file systems [8, 11, 26, 28, 35]. MetaSync uses content-based addressing for a unique purpose, allowing us to asynchronously uploading or downloading objects to backend services. While algorithms for distributing or replicating objects have also been proposed and explored by past systems [10, 33, 34], the replication system in MetaSync is designed to minimize the cost of reconfiguration to add or subtract a storage service and also to respect the diverse space restrictions of multiple backends.

7 Conclusion

MetaSync provides a secure, reliable, and performant file synchronization service on top of popular cloud storage providers. By combining multiple existing services, it enables a highly available service during the outage or even shutdown of a service provider. To achieve a consistent update among cloud services, we devised a client-based Paxos, called pPaxos, that can be implemented without modifying any existing APIs. To minimize the redistribution of replicated files upon a reconfiguration of services, we developed a deterministic, stable replication scheme that requires minimal amount of shared information among services (e.g., configuration). MetaSync supports five commercial storage backends (in current open source version), and outperforms the fastest individual service in synchronization and cloning, by 1.2-10× on our benchmarks. MetaSync is publicly available for download and use (`http://uwnetworkslab.github.io/metasync/`).

Acknowledgments

We gratefully acknowledge our shepherd Feng Qin and the anonymous reviewers. This work was supported by the National Science Foundation (CNS-0963754, 1318396, and 1420703) and Google. This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0107. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Taesoo Kim was partly supported by MSIP/IITP [B0101-15-0644].

References

[1] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A case for cloud storage diversity. In *Proceedings of ACM Symposium on Cloud Computing (SoCC)*, 2010.

[2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, Jon, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, 2002.

[3] C. Ball. Announcing gittorrent: A decentralized github. `http://blog.printf.net/articles/2015/05/29/announcing-gittorrent-a-decentralized-github/`, 2015.

[4] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of ACM EuroSys conference*, pages 31–46, 2011.

[5] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo. SCFS: A shared cloud-backed file system. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 169–180, 2014.

[6] C. Brooks. Cloud Storage Often Results in Data Loss. `http://www.businessnewsdaily.com/1543-cloud-data-storage-problems.html`, October 2011.

[7] S. Byrne. Microsoft OneDrive for business modifies files as it syncs. `http://www.myce.com/news/microsoft-onedrive-for-business-modifies-files-as-it-syncs-71168`, Apr. 2014.

[8] 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. u. Haq, M. I. u. 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 *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 143–157, 2011.

[9] Canonical Ltd. Ubuntu One: Shutdown notice. `https://one.ubuntu.com/services/shutdown`.

[10] A. Cidon, S. M. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (ATC)*, pages 37–48, 2013.

[11] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in SAN cluster file systems. In *Proceedings of the 2009 USENIX Conference on Annual Technical Conference (ATC)*, 2009.

[12] J. Cook. All the different ways that 'icloud' naked celebrity photo leak might have happened. `http://www.businessinsider.com/icloud-naked-celebrity-photo-leak-2014-9`, Sept. 2014.

[13] J. Ćurn. How a bug in dropbox permanently deleted my 8000 photos. `http://paranoia.dubfire.net/2011/04/how-dropbox-sacrifices-user-privacy-for.html`, 2014.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[15] Dropbox. Thanks for helping us grow. `https://blog.dropbox.com/2014/05/thanks-for-helping-us-grow/`, May 2014.

[16] dropbox-api. Dropbox API. `https://www.dropbox.com/static/developers/dropbox-python-sdk-1.6-docs/index.html`, Apr. 2014.

[17] 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 (OSDI)*, 2010.

[18] A. J. Feldman, A. Blankstein, M. J. Freedman, and E. W. Felten. Social networking with Frientegrity: Privacy and integrity with an untrusted provider. In *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012.

[19] E. Gafni and L. Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, Feb. 2003.

[20] git. Git Internals - Git Objects. `http://git-scm.com/book/en/Git-Internals-Git-Objects`.

[21] google-api. Google Drive API. https://developers.google.com/drive/v2/reference/, Apr. 2014.

[22] R. Gracia-Tinedo, M. S. Artigas, and P. G. López. Cloud-as-a-Gift: Effectively exploiting personal cloud free accounts via REST APIs. In *IEEE 6th International Conference on Cloud Computing (CLOUD)*, 2013.

[23] G. Huntley. Dropbox confirms that a bug within selective sync may have caused data loss. https://news.ycombinator.com/item?id=8440985, Oct. 2014.

[24] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC)*, pages 654–663. ACM, 1997.

[25] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[26] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–9, 2004.

[27] 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 (OSDI)*, pages 1–12, 2010.

[28] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazières. Replication, history, and grafting in the Ori file system. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP)*, pages 151–166, 2013.

[29] R. C. Merkle. A digital signature based on a conventional encryption function. In *In Proceedings of the 7th Annual International Cryptology Conference (CRYPTO)*, pages 369–378, Santa Barbara, CA, 1987.

[30] E. Mill. Dropbox Bug Can Permanently Lose Your Files . https://konklone.com/post/dropbox-bug-can-permanently-lose-your-files, October 2012.

[31] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *USENIX Security*, 2011.

[32] J. Nelson and L. Peterson. Syndicate: Democratizing cloud storage and caching through service composition. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, pages 46:1–46:2, 2013.

[33] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–15, 2012.

[34] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, 1988.

[35] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[36] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 149–160, 2001. ISBN 1-58113-411-8.

[37] D. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, 1998.

[38] Z. Whittaker. Dropbox under fire for 'DMCA takedown' of personal folders, but fears are vastly overblown. http://www.zdnet.com/dropbox-under-fire-for-dmca-takedown-7000027855, Mar. 2014.

[39] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 292–308, 2013.