MetaSync: File Synchronization Across Multiple Untrusted Storage Services

Seungyeop Han, Haichen Shen, Taesoo Kim[†], Arvind Krishnamurthy, Thomas Anderson, and David Wetherall
University of Washington, [†]MIT CSAIL
University of Washington Technical Report UW-CSE-14-05-02

Abstract

Cloud-based file synchronization services, such as Dropbox and OneDrive, are a worldwide resource for many millions of users. However, individual services often have tight resource limits, varying performance in regions of the world, 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 linearizable updates on top of the unmodified APIs exported by existing services. Our system automatically redistributes files upon adding, removing, or resizing a provider with a novel deterministic replication scheme.

Our evaluation shows MetaSync provides low update latency and high update throughput, close to the performance of commercial services, but is more reliable and available. For synchronization, MetaSync outperforms its underlying cloud services by 1.2X-10X on three realistic workloads.

1 Introduction

Cloud-based file synchronization services have become tremendously popular. Dropbox reached 200M users in November 2013, doubling its customer base over the previous year [8]. Many competing providers offer similar services, including Google Drive, Microsoft OneDrive, Box, and Baidu in China. With such resources, mostly for free, users are likely to upload ever larger amounts of personal and private data.

Unfortunately, not all services are trustworthy or reliable. Storage services routinely lose data due to internal faults [2], and can leak users' personal data [22]. They may block access to content, e.g., DMCA takedowns [29] to comply with the law. From time to time, entire clouds may go out of business, e.g., Ubuntu One [5]. Some storage services, e.g., OneDrive [3], even intentionally alter private user data in violation of the basic storage contract.

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, data can be encrypted by a user before being stored in the cloud for confidentiality. More generally, research such as Depot [19] and SUNDR [18] shows how to design systems from scratch in which users of the cloud storage obtain data confidentiality, integrity, or availability without trusting the underlying storage provider.

Our work differs from prior work in two ways. First, we build a synchronization service on top of existing cloud storage (Figure 1) to leverage resources that are mostly well-provisioned and managed, normally reliable, and inexpensive (free!). This is challenging because we must work correctly using only their unmodified APIs. Second, we combine multiple providers not only for larger storage space, but also for a system that is more reliable (via replication), consistent (via our client-driven Paxos protocol), and faster (via splitting or combining file objects optimized for current services). 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 contributions, beyond MetaSync itself as a ready-to-use open source project, are twofold: 1) pPaxos, a client-based Paxos algorithm that provides a consistent global state across multiple (and passive) storage backends, and 2) a novel deterministic replication algorithm that provides stable mapping schemes for file objects upon re-configurations of services, such as increasing capacity or adding/removing a service. MetaSync strictly maintains global consistency among backend services, assuming a user's computers function correctly.

The rest of this paper is organized as follows. We state our goals and threat model (§2), and explain our design (§3). We describe our implementation (§4), evaluate (§5). Finally, we compare MetaSync to related work

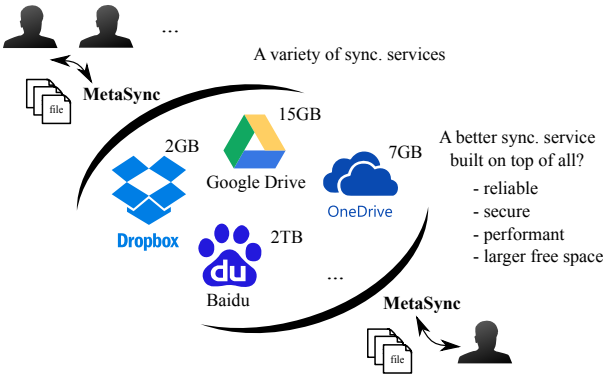

Figure 1: Overview of MetaSync for end-users. MetaSync provides reliable, secure, and performant synchronization services on top of untrusted commercial services, along with larger storage space for free.

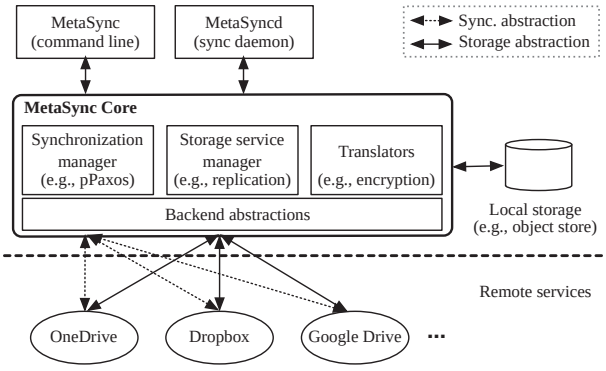

Figure 2: Overview of design. MetaSync has three main components: storage service manager to coordinate replication; synchronization manager to orchestrate cloud services; and translators to support data encryption and integrity. The components are implemented on top of an abstract cloud storage API, which provides a uniform interface to storage backends such as Dropbox and Google Drive. MetaSync supports two front-end interfaces: a command line interface for users and a synchronization daemon for automatic monitoring and check-in.

(§6) and conclude (§7).

2 Goals and Assumptions

The usage model of MetaSync matches that of existing synchronization services such as Dropbox. The 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. Local updates are automatically reflected to all connected hosts; conflicting updates are flagged for manual resolution.

For users desiring explicit control over the merge process, we also provide a manual git-like push/pull interface. In this case, the user creates a set of updates and runs a script to apply the set atomically. The system accepts an update only if it has been merged with the latest version pushed by any user. Any user can pull the latest version, or, as with git, the user can revert to any previously committed version.

In terms of security, we assume that the backend services are curious, incompetent, but not actively malicious. The storage services may try to discover which files are stored by which user along with the content of the files, and they may accidentally corrupt or delete files. However, we assume that services do not collude, 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. Finally, we assume that clients sharing directories and running MetaSync are trusted.

With this threat model, the goals of MetaSync are:

- **Availability:** User files are always available for both read and update despite any predefined number of service outages, even if a provider completely stops providing any access to its previously stored data.
- **Integrity:** Any corruption of files should be detectable provided that a majority of services do not collude.
- **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.
- **Performance:** The system should benefit from the combined capacity of the underlying services. We aim to provide faster synchronization service than any individual service.
- **Reconfiguration:** Adding a service into our system does not require significant extra work.
- **No direct client-client communication:** All clients should be able to synchronize without any direct communication among them. In particular, they never need to be online at the same time.

3 System Design

This section describes the design of MetaSync as illustrated by Figure 2. The core library defines abstractions for cloud storage services, and all components are implemented on top of those abstractions, making it easy to incorporate a new storage service into the system (§3.5). MetaSync consists of three major components: storage service manager, synchronization manager, and translators. The storage service manager maintains the replication of file objects by mapping those objects to storage services in a deterministic way, making our system resilient to tear-down of services and allows for user's reconfiguration of services (§3.2). The synchronization manager provides support to make sure that every client has a consistent view of the user's synchronized files, by orchestrating storage services with pPaxos (§3.3). The translators implement optional modules for encryption and decryption of file objects in services, and these mod-

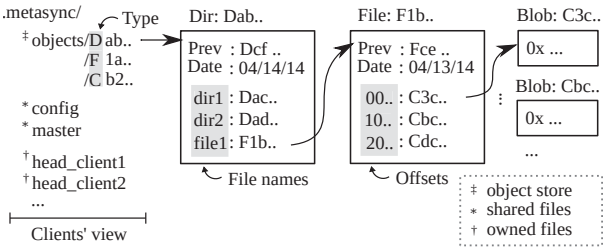

Figure 3: File management in a client's local directory. The object store maintains user files and directories with content-based addressing, in which the name of each object is based on the hash of its content. Each client also maintains two kinds of files: shared, which all clients update; and owned, for which the owner client is the only writer. Therefore, while the object store and owned files can be updated without synchronization, updates to the shared files require coordinated updates of the backend stores; this is done by the synchronization manager (§3.3).

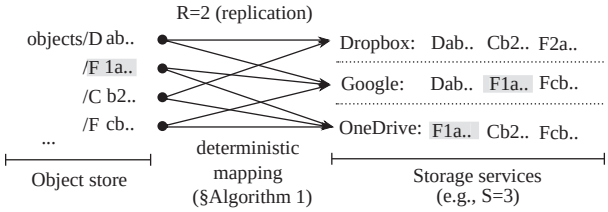

Figure 4: Example of replicating objects ($R = 2$) to the backend storage services ($S = 3$). Each object deterministically maps into a group of storage services for replications (Figure 5).

ules can be transparently composed to enable flexible extensions (§3.6).

3.1 File Management

MetaSync has a similar underlying file structure to that of git [14] in managing files and their versions: objects, units of storing files, are identified by the hash of their content to avoid redundant use of storage; directories form hash trees, similar to Merkle trees [21], where the root directory's hash is the root of the tree. Unlike git, MetaSync divides and stores each file into chunks, called Blob objects, 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 (annotated with an object type as a prefix in Figure 3). A File object contains hash values of Blob objects and their offsets. A Dir object contains hash values of File objects and their names. File and Dir objects keep track of hash chains of their old versions as well (Prev in Figure 3).

Each client also maintains two kinds of metafiles to provide a consistent view of the global state: *shared files*, which all clients can update; and *owned files*, which only the single owner (writer) client can update.

Shared files. MetaSync has two shared files: `config` and `master`. `config` keeps the configuration of backend services like capacities, authenticators, and encryption keys. `master` keeps the hash value of the root directory, representing the consistent view of the global state. When updating these shared files, we invoke a synchronization protocol built from the APIs provided by existing cloud storage providers (as described in §3.3).

Owned files. Unlike shared files, updating owned files does not require a synchronization protocol across storage services (by design). For example, a client can check-in a file to the object store, upload to the backend services, and update its local head (`head_client_*`) without requiring any coordinated updates to the backend storage providers.

For initiation, a user sets up a directory to be managed by MetaSync; files and directories under the directory should be synchronized. This is equivalent to a repository in version control systems. Then, MetaSync creates a directory containing the files as shown in Figure 3 and starts synchronization over backend services based on user configuration. Each managed directory has a name (called namespace) in the system to be used in synchronizing with other clients. MetaSync creates a folder with the name in each backend. The folder at the backend storage stores the same set of files as clients, along with a subset of objects based on the mapping we explain next. A user can maintain multiple directories having different configuration and composition of backends to synchronize only necessary files for each client.

3.2 Replication

MetaSync replicates objects (in the object store) redundantly across R storage providers (typically $R = 2$) to provide higher availability even when a service is temporarily inaccessible. This also provides potentially better performance over wide area networks. However replication comes at the cost of maintaining shared information regarding the mapping of objects to services. In our settings, where the storage services passively participate in the coordination protocol, providing a consistent view of this shared information is particularly expensive. 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 storage space they provide, ranging from 2GB in Dropbox to 2TB in Baidu.

Goals. We desire an efficient mapping scheme that maps each object to a group of services over which it is replicated. Given a hash of an object (modulo H), the mapping should return a replication set, as indicated below:

$$map : H \rightarrow \{s : |s| = R, s \subset S\}$$

where

- H is the hash space.
- S is the set of services.
- R is the number of replicas.

The mapping scheme in MetaSync has to satisfy following requirements:

R1 Share minimal information amongst services.

R2 Support variation in storage size limits across different services and across different users.

R3 Minimize realignment of objects upon removal or addition of a service. (Though we assume that this would happen rarely, the impact should be minimal when it happens.)

One option to perform this mapping is to use consistent hashing, which maps both storage providers and objects onto an identifier circle and then assigns objects to storage providers based on their positions on the identifier circle. While this mechanism minimizes realignment of objects upon service changes and supports arbitrarily large numbers of objects and storage providers, it does not support user-specific storage limits at different providers nor does it achieve optimal or tight bounds on load balance across different providers [27]. Another option is to maintain a mapping at the granularity of individual objects. For example, it is possible to build a mapping table that tracks which services store an object and share this table amongst clients sharing a set of directories. However, it is expensive to share such a mapping as the table can be large and is frequently updated (which is against R1). Thus, instead of building a table, we devise a deterministic mapping which can be reconstructed from a small amount of shared information. Our scheme is similar in spirit to consistent hashing [16] (R3), but modified not to rely on random positioning of the nodes. Our deterministic mapping scheme is unique in that it can reflect the space limits at each service (R2). The H parameter can be used to adjust the extent to which we satisfy R2—a bigger value for H is likely to produce a better-balanced mapping that achieves storage utilization in proportion to the imposed storage limits.

Algorithm. Our deterministic mapping scheme is formally described in Figure 5. For each backend storage provider, the mapper 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 [9].) 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, 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 [28]. These are then the storage services over which MetaSync replicates the object.

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

Figure 5: The deterministic mapping algorithm.

Note that this 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 users sharing a set of files. The list of services and the capacity limits (see S in Figure 6) is part of the service configuration and is shared through the `config` file. The virtual node list is populated proportionally to service capacity, and the ordering inside each list is determined by a uniform hash function. Thus, the resulting mapping of objects onto services should be proportional to service capacity limits for large values of H (R2 holds). Finally, 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 6 illustrates an example of how our mapping scheme works 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 as in 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). The gray-marked services in the replication set indicates the realignment of objects producing the same hash value in the hash space.

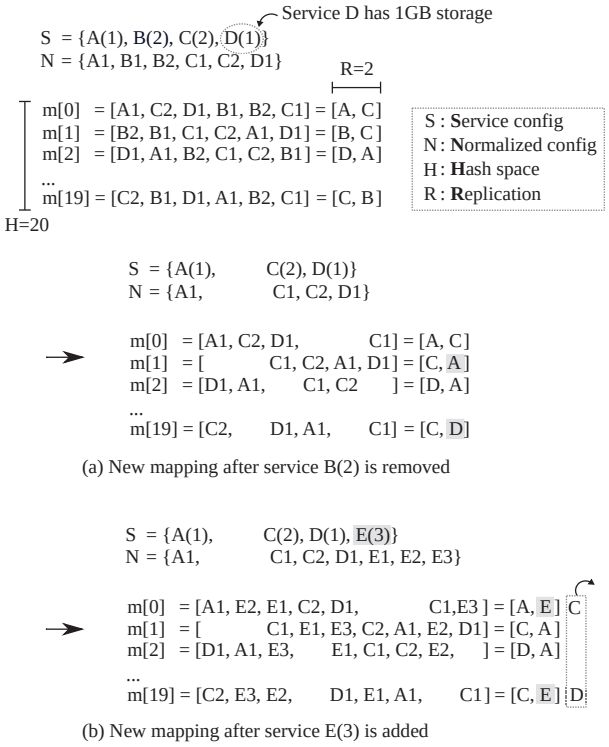

Figure 6: An example of deterministic mapping and its reconfigurations, where a service is removed in (a) and then a service is newly added in (b). The initial mapping is deterministically generated by Figure 5, given the configuration of four services, $A(1), B(2), C(2), D(1)$ where the number in a parenthesis represents the storage capacity of each service. (a) shows a new mapping after service B is removed from the initial service configuration, and (b) shows a new mapping after service E (with 3GB) is added after (a). The grayed mappings indicate the new replication upon reconfiguration, and the dotted rectangle in (b) represents redundant replications that will be garbage collected later.

3.3 Synchronization: pPaxos

The file structure allows MetaSync to minimize synchronization barriers. Each object in the object store can be independently uploaded as it uses content-based addressing. Each owned file (e.g., `head_client_*`) is also independent since we ensure that only the owner client modifies the file. Thus, a potential race condition can only occur when a client wants to modify the shared files (e.g., `master`).

Challenges. In distributed settings with multiple clients and storage providers, it is not straightforward to create a synchronization barrier to coordinate updates. This is particularly the case for MetaSync, where we assume that backend storage services are passive, meaning that protocols for synchronization are performed only by clients, and direct communication between clients are not allowed. Clients only communicate indirectly through storage providers. Furthermore, a client should be able to make progress even when some services are down or slow.

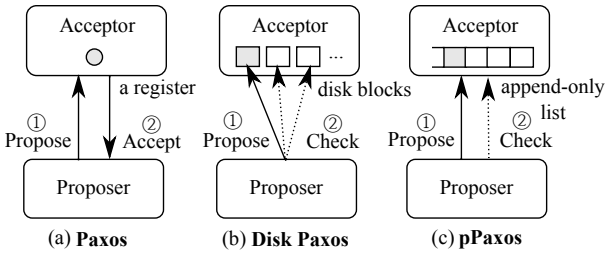

Figure 7: Comparison of operations between a proposer and an acceptor in Paxos [17], Disk Paxos [13], and pPaxos. When proposed, an acceptor in Paxos makes a decision and sends it to the proposer, whereas proposed data is stored in per-client disk blocks in Disk Paxos and in an append-only list in pPaxos. In Disk Paxos, the proposer needs to check a block for every client to determine which proposal was accepted. In comparison, Paxos can be considered as having a register to store the proposal.

pPaxos. Creating a synchronization barrier for shared files can be reduced to the problem of having the clients come to a consensus on which client can perform the next update to the shared files. Since clients do not have communication channels between each other, they need to rely on storage services to achieve this consensus. We devise a variant of Paxos [17], called pPaxos (passive Paxos). Using pPaxos, a client proposes that it will change the shared files and gets a lease for performing the update when the proposal is accepted.

We overview pPaxos by relating it to the classic Paxos algorithm (see Figure 7(a)). Each client works as a proposer and learner, and it relies on backend services as acceptors. The major challenge here is that we cannot assume that the backend services will implement the Paxos acceptor algorithm and provide the corresponding APIs. Instead, we require them to provide a simple storage API corresponding to that of an *append-only list* (Figure 7(c)). The append-only list *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, as all storage services we examine linearize the order of API invocations. With this append-only list abstraction, backend services can act as *passive acceptors*. While these acceptors cannot actively decide as to which proposal is promised or accepted, clients who retrieve a set of messages stored on the list can determine the "accepted" decisions, under the assumption that other clients follow the pPaxos protocol as well. This is why we refer to the storage providers as passive acceptors; acceptors delegate the decisions to clients and only respond with what they have stored on the append-only list.

We note that this is possible due to the nature of backend storage APIs. With a log, all clients see the

Proposer

```
1:  procedure PROPOSEROUND(value, round, acceptors)
    prepare:
2:      concurrently
3:      for all a ← acceptors do
4:          SEND(⟨PREPARE, round⟩ → a)
5:          newlog ← FETCHLOG(a)
6:          CHECKIFACCEPTED(a, newlog)
7:          if a.round > round then abort
8:      wait until done by a majority of acceptors
    accept:
9:      accepted ← {a.accepted | a ∈ acceptors}
10:     if |accepted| > 0 then
11:         p ← arg max{p.round | p ∈ accepted}
12:         value ← p.value
13:     proposal ← ⟨round, value⟩
14:     concurrently
15:     for all a ← acceptors do
16:         SEND(⟨ACCEPT, proposal⟩ → a)
17:         newlog ← FETCHLOG(a)
18:         CHECKIFACCEPTED(a, newlog)
19:         if a.round > round then abort
20:     wait until done by a majority of acceptors
    done:
21:     return proposal
22: procedure CHECKIFACCEPTED(acc, logs)
23:     for all l ∈ logs do
24:         switch l do
25:             case ⟨PREPARE, round⟩
26:                 acc.round ← max(l.round, acc.round)
27:             case ⟨ACCEPT, prop⟩
28:                 if l.prop.round ≥ acc.round and
29:                  l.prop.round > acc.accepted.round then
30:                     acc.accepted ← l.prop
31: procedure ONRESTARTAFTERFAILURE(round)
32:     round ← round + 1
33:     WAITEXPONENTAILLY
34:     PROPOSEROUND(value, round, acceptors)
```

Passive Acceptor

```
35: procedure ONNEWMESSAGE(⟨msg, round⟩)
36:     APPEND(⟨msg, round⟩ → log)
```

Figure 8: pPaxos Algorithm.

same order; thus they come to the same conclusion as to which proposal is accepted. Since the underlying APIs vary across services, we summarize the details of how MetaSync provides the append-only list abstraction for each provider in Table 3. Note that the set of pPaxos acceptors need not be the same as the set of storage service backends. As future work, MetaSync can even use non-storage services like Twitter as pPaxos acceptors in order to provide the necessary synchronization operations.

Algorithm. With the availability of an append-only list, the algorithm itself becomes a simple adaptation of the classic Paxos, but one where the decision making is performed by proposers. It mainly replaces acceptors' responses by client actions to fetch the logs and make decisions based on the contents of the logs. As Figure 8 shows, 1) when a client wants to acquire a lease, it sends a `PREPARE` to all the acceptors with a round number (Line 3-4). Then, it fetches the logs from the acceptors and checks whether the round number is promised by acceptors (Line 5-6, 25-26). It aborts this round of the proposal if it sees an acceptor who has already promised a round number which is larger than its current round number (Line 7). 2) If there are any accepted proposals, it proposes the accepted proposal with the largest proposal number to the acceptors; otherwise, it proposes the client itself as the lease holder for the current round number by sending `ACCEPT_REQ` to all acceptors (Line 9-16). The `ACCEPT_REQ` will be accepted when a majority of acceptors have a smaller or equal round number (Line 17-20, 27-30). When it is accepted by the quorum, it can safely commit that it has the lease (Line 21). In case it fails, it does random exponential back-off and tries again (Line 31-34).

The lease is expired after a timeout or when the holder releases it. If the holder wants to keep it beyond expiration, it needs to extend the timeout by running the pPaxos algorithm again. However, we assume that each client does not require the lease to be long as updating `master` takes a short amount of time, given that the shared file is just a pointer to the root object. When the holder finishes its modifications, it sends a `DONE` message to each acceptor.

Note that this setting and the following algorithm is similar to that of Disk Paxos [13], and pPaxos can be considered as an optimized version of Disk Paxos ((Figure 7(b)) as it uses fewer number of messages by taking advantage of the append-only list abstraction.

pPaxos in action. When files in the system are changed, an update happens as follows (see Figure 9): 1) the client updates the local objects and `head_client` to point to the current root. Again, note that both updates do not require running pPaxos, 2) put the data blocks to the appropriate backend services, and 3) it updates master with its `head_client`. As mentioned, the last operation requires MetaSync to run pPaxos and get a lease as only one client can modify the `master` file at a time. If it fails to get the lease, it needs to wait until the lease holder finishes updating or the corresponding lease times out. When it succeeds, it updates `master`, and stores it on all connected backends. Note that it may update the quorum ($\lceil n/2 \rceil$) first, and then update the rest in background. Then, clients polling the `master` are notified, and they fetch new objects from backend services in parallel,

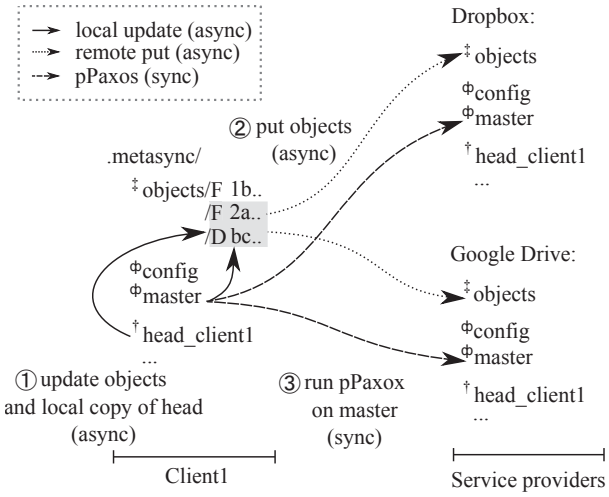

Figure 9: pPaxos in action: a file check-in. ① MetaSync converts the file to an object, and updates its local copy of head to point to the newly-updated root directory (§3.1). ② Then, MetaSync asynchronously puts new objects redundantly into backend services, based on our mapping scheme (§3.2). ③ Finally, MetaSync runs pPaxos to update `master`, providing a consistent view to the global state among clients accessing multiple storage backends. Note that it may run garbage collection later to remove unused objects (§3.8).

based on their replication sets. pPaxos may also be used for proposing values of the shared files directly, instead of proposing a lease holder.

Merging. Merging can happen under the following conditions: (1) when a client synchronizes its local head with the `master`; (2) when a client wants to update the `master`. More specifically for the second condition, before requesting a lease to update the `master`, the client must merge the latest version of `master` into its `head_client`. After acquiring the lease, if the latest version has been updated by others, the client needs to release the lease and retry merging with the latest version.

For merging `master` into `head_client`, MetaSync employs three-way merging as in other version control systems. It allows many conflicts to be automatically resolved. Three-way merging cannot resolve all the conflicts, as two clients may change the same parts of a file. In that case, it can delegate applications to make a decision. In our current implementation of sync daemon, for example, it generates a new version of the file with `.conflict.N` extension, which allows for the users to resolve it later.

3.4 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 be designed to tolerate faults. MetaSync achieves fault-tolerance via replication (§3.2) for data and via pPaxos for consistency control (§3.3).

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
share(`path`,`email`)	Share `path` with `email`
(b) Synchronization abstraction	
append(`path`, `msg`)	Append `msg` to the list at `path`
fetch(`path`, `index`)	Fetch a log at `index` from `path`

Table 1: Abstractions for backend storage services.

Data model. By replicating each object into multiple backends (R in §3.2), 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 [14], the hash tree ensures each object's hash value is valid from the root (`master`). Then, each object's integrity can be verified by calculating the hash of the content and comparing with the name. The `master` file 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 files, `config` and `master`. The underlying pPaxos protocol requires $2f+1$ acceptors to ensure correctness if f nodes may fail under the fail-stop model. The shared files are stored across all the backends. Assuming non-byzantine behavior, the shared files can be stored in $f+1$ backends, and clients take the most recent version by comparing the history.

3.5 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 1(a). Many storage services provide a developer toolkit to build a customized client accessing user files [10, 15]; 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 implements storage backends with many different 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 1(b). How to utilize the underlying APIs to build the append-only log varies across services. Note that the set of services for synchronization abstraction does not need to be the same with storage service backends. We summarize how MetaSync builds it for each provider in Table 3.

3.6 Translators

MetaSync provides a plugin system, called Translators, for encryption. Translators is highly modular so can easily be extended to support a variety of other transformations such as compression. Plugins in Translators should implement two interfaces, `put` and `get`, which will be invoked before storing to and after retrieving 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.

3.7 Sharing

Sharing a folder for collaboration is one of the important features in many synchronization services. As backend services support 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. The person who initiated sharing may also stop sharing similarly. 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.

3.8 Other Issues

Collapsing directory All storage services manage individual files for uploading and downloading. As we see later in Table 5, throughput for uploading and downloading small files are very low compared to those for larger files. As an optimization, we collapse all files in a directory into a single object when the total file size is small enough.

Garbage collection Each object is immutable, hence modifying a file creates new objects and leaves the old objects associated with the file obsoleted. To prevent waste of space, we must perform garbage collection periodically. A client doing garbage collection first retrieves each client's head from the backends. If there are distinct head files for a client, it finds the most up-to-date version. Traversing through tress from the head files and the master, objects not appearing in any client's tree can be safely removed. Note that when user wants to keep old versions, they can create a snapshot by storing a root pointer of the snapshot, and objects used in the snapshots would not be garbage collected.

Component	Lines of code
Synchronization Manager	320
Storage service	4,512
Translators	78
Mapping scheme	259
Etc	2,354
Total	7,523

Table 2: Components of our MetaSync prototype, and their estimated complexity, in terms of lines of Python code.

Versioning. MetaSync keeps track of versions of entire objects by default. It allows user to roll back to any history version by checking out a particular head. Besides, users can also custmoize the length of history as a parameter to the garbage collection. Note that some storage services already provide some form of versioning (or revision) features. By integrating these existing features, MetaSync can not only remove wasted storage for keeping old versions in the object store, but also improve the performance of the garbage collection.

4 Implementation

We have implemented a prototype of MetaSync in Python, components of which are summarized in Table 2. 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. All storage services provide APIs equivalent to MetaSync's storage abstractions, like `get()` and `put()`, defined in Table 1. Since each service varies in supporting `poll()`, 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 in Baidu on Table 3.

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 mod-

Service	Synchronization API		Storage API
	`append()`	`fetch()`	`poll()`
Box Google OneDrive	Create an empty `log` file and post *comments* to the `log` file.	Download the entire comments attached on the `log` file. To reduce the overhead of downloading the entire log, obsoleted comments are deleted during a garbage collection.	Use `events` API, allowing long polling. But it monitors over all files rather than a specific directory. (Google and OneDrive: periodically get `master` and see if any changes since the last fetch.)
Baidu	Create a `log` directory, and consider each file as a log entry. To order each log entry (files), we assign a monotonically increasing sequence number to each file name. If the number is already taken, we will get an `ItemAlreadyExists` error and try with a next sequence number.	List the `log` 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. But it monitors all files in the account rather than a specific directory.
Dropbox	Create a `log` file, and overwrite the file with a new log entry, relying on Dropbox's versioning (revisions).	Request a list of versions (revision history) of the `log` file.	Use `longpoll_delta`, a blocked call, that returns if there is a change under `path`.
Disk†	Create a `log` file, and append a new log entry at the end of the file.	Read the new log entries from the `log` 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).

ified 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`, `pull`).

5 Evaluation

To evaluate MetaSync, this section answers the following questions in turn:

- 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 gigabit LAN except for synchronization performance in §5.4. Since most of services do not have native clients for Linux, we compared synchronization time for native clients and MetaSync on Windows desktops connected to 100Mbps LAN.

5.1 Design constraints

We first estimate the economic value of MetaSync's free space, combining five commercial synchronization services (see Table 4). The free space (2082 GB used in this experiment) MetaSync provides is worth $750 of Amazon S3. Users can also add more free or commercial services, or even multiple accounts in the same service. In addition to the economic value of the amount of storage,

Service	Free space (GB)	Cost ($/GB/year)
Amazon S3	-	$0.36
Box	10 GB	$0.60
Baidu	2048 GB	$0.80
Dropbox	2 GB	$1.20
Google Drive	15 GB	$0.24
OneDrive	7 GB	$0.50

Table 4: Amount of free space provided by each service and costs for additional space. For cost we listed cheapest prices per GB and year.

MetaSync's goal is to build a reliable and performant service on top of potentially fragile backend services. We measured the performance variance of commercial services in Table 5. One important observation is that all services are slow in handling small files. This provides MetaSync the opportunity to beat their performance by combining small objects.

5.2 pPaxos performance

We design a micro-benchmark to measure how quickly pPaxos reaches the consensus, varied on the number of concurrent writers (clients). The results of the experiment with 1-5 clients and single passive acceptor over 5 storage providers are summarized in Figure 10. A single run of pPaxos took about 3.6 sec on average under a single writer model to verify acceptance of the proposal. This requires four round trips. It took about 6.5 sec with 5 competing clients. 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. Although the fastest client would likely win each round

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 5: 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.

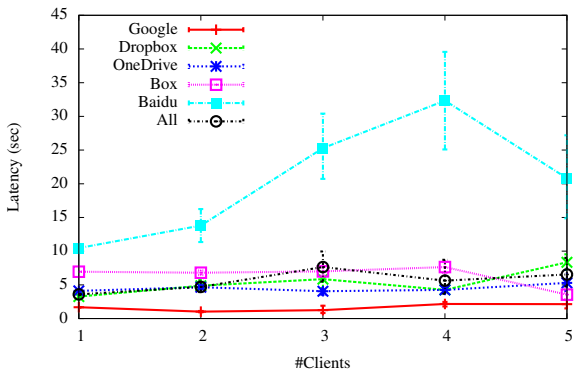

Figure 10: Latency (sec) to run a single pPaxos round with combinations of backend services and competing clients: when using 5 different storage providers as backend nodes (all), the common path of pPaxos at a single client takes 3.6 sec, and the slow path with 5 competing clients takes 6.5 sec on average.

(unfair), we believe this situation is not a problem at all in practice. We also measured the latency of running pPaxos over multiple storage providers. Figure 10 shows the result of 1-5 clients runs over all 5 storage providers. Compared with results of the single storage provider, the latency doesn't degrade with increase of the number of storage providers.

5.3 Deterministic mapping

We then evaluate how fairly our deterministic mapping distributes objects into storage services with different capacity requirements, in three replication settings ($R = 1,2,3$). We tested 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 6. 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 objects will be allocated into Baidu, to satisfy the storage requirement of each service. This situation improves for $R = 2$, since objects will be placed into other services beyond Baidu where a large number of copies are already stored. Baidu gets 6.2 MB of more storage when increasing $R = 2 \rightarrow 3$, and our mapping scheme preserves the capacity requirements for the rest of services (using 1.3%). Even for the challenging case, $S = 5, R = 3$ where an object should be stored in more than majority of services, MetaSync's mapping scheme produces distribution of objects that uses close to even fraction of each storage, yet deterministic and resilient to reconfiguration, which we cover next.

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 after calculating the mapping once it is stored locally as a cache, which is packed as 3MB.

The relocation scheme is resilient to its changes as well, meaning that redistribution of objects should be 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 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 garbage collect 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 on idle time.

5.4 End-to-end performance

We selected three kinds of workloads to demonstrate performance characteristics of MetaSync. 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).

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%)
$R=3$	13,039 (4.93%) 67.2 MB (3.28%)	66,964 (25.33%) 355.7 MB (2.32%)	54,505 (20.62%) 294.8 MB (2.88%)	41,752 (15.79%) 222.7 MB (3.11%)	88,130 (33.33%) 470.1 MB (0.02%)	264,390 (100%) 1410.4 MB (0.07%)

Table 6: Replication results generated by our deterministic mapping scheme (§3.2) for Linux kernel 3.10.38 (Table 8) on 5 different services with various storage space, given for free. We synchronized total 470 MB of files, consisting of 88k objects, and replicated ($R=2, R=3$) 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, and $R=3$ provides the strongest fault-tolerance ($f=2$), resilient against simultaneous failures of two services.

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 8: Synchronization performance (sec) 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 two times, MetaSync outperform $>$10 times faster than Dropbox in Linux kernel and 2.3 times faster in photo sharing; we can finish the synchronization right after uploading a single replication set (but complete copy) and the rest replication will be scheduled in background. To understand how slow straggler (Baidu) affects MetaSync's performance ($R=1$), we also measured synchronization time on $S=4$ without Baidu, where MetaSync vastly outperforms all of commodity services

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 7: Time to relocate 193 MB amount of objects (photo-sharing workloads in Table 8) 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.

Table 8 summarizes our experimental results of end-to-end synchronization performance for all workloads, comparing native clients provided by each service with MetaSync. Each workload was copied into one client's synchronized directory before synchronization started. The synchronization time was measured as the length of interval between one desktop starts uploading 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 four different settings. MetaSync outperforms any individual service for all three 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 those clients from the services 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 to note, 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 on background.

Clone. Storage services often limit its throughput of downloading: for example, Dropbox is saturated its bandwidth at 5.1 MB/s and Google Drive is similarly saturated at 3.4 MB/s, shown in Figure 11. By using multiple storage services, MetaSync can fully exploit the bandwidth of local connection of users, not limited by the allowed throughput of each services. For example, if Dropbox took 38.0 sec and Google Drive took 57.2 sec to download 193 MB data, then the ideal download speed of using both can be computed, $1/(1/38.0+1/57.2) = 22.8$. In comparison, MetaSync with (S=2, R=2) took 25.5 sec, which we believe it approximately reaches to

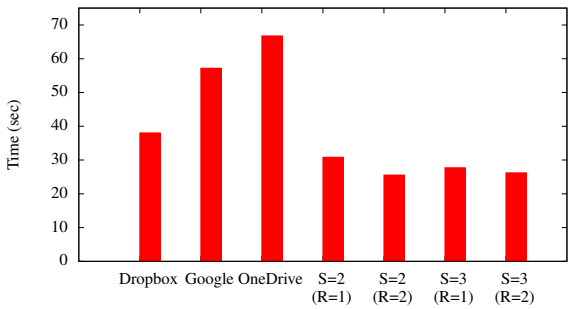

Figure 11: Time (sec) to clone an entire storage of 193 MB photos. When using individual services, Dropbox, Google, and OneDrive, it took 40-70 sec to clone, but MetaSync could improve the performance of cloning, 25-30 sec (30%) by leveraging the distributions of objects across multiple services.

the theoretical limit, considering all the extra bookkeeping in MetaSync.

6 Related Work

MetaSync borrows many ideas from prior systems. We maintain file objects similar to a distributed version control system like [14], or Ori file system [20], but MetaSync can combine or split each file object for efficient store and retrieval. Content-based addressing is not a new idea, used by existing file systems [4, 18, 20, 26], or by deduplication [7]. However, MetaSync utilized content-based addressing for a unique purpose, asynchronously uploading or downloading objects to backend services. While algorithms for distributing or replicating objects have also been proposed and explored by past systems [6, 24, 25], MetaSync devised our replication scheme for deterministic way to minimize sharing, and also to satisfy space restrictions of multiple backends.

A major line of work, starting with SUNDR [18] but carrying through SPORC [12], Frientegrity [11], and Depot [19], is how to provide tamper resistance and privacy on untrusted storage nodes. These systems develop various methods of detecting and resolving equivocations after the fact, but ultimately they have 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. We make 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.

Syndicate [23] provides applications with a storage layer by composing existing storage systems. Compared to MetaSync, it is designed as storage APIs for applications rather than synchronization services for end users. Thus, they mostly delegate design choices like how to manage files and replicate to application policy. Furthermore it needs to run a separate metadata service. Recent studies propose combining multiple cloud services as storage backend [1, 30] for the goal such as minimizing cost and provide better reliability. These systems rely on server-side solutions as well.

The coupling between user's local disk and cloud storage may cause the data loss and inconsistency in the cloud due to the local data corruption and crashes during synchronization. Even worse, such data corruption may pollute all copies on other devices. ViewBox [31] detects corrupt data through data checksumming and ensures the consistency by adopting view-based synchronization. MetaSync can also guarantee data integrity through the hash-based file objects and provide linearizable updates by using pPaxos.

7 Conclusion

MetaSync provides a secure and reliable file synchronization service on top of cloud storage providers in which the user trusts only open-source software running on their own computers. By combining multiple existing services, it provides a highly available service during the outage or even shutdown of a provider. To achieve a consistent update among cloud services without modifying their APIs, we devised a client-based form of Paxos. To let users easily resize the storage of services, add a new service or move out from a service, we developed a replication scheme that deterministically duplicates and redistributes file objects with minimal sharing among services. We prototyped MetaSync with five commercial storage backends, and our benchmark shows it outperforms the fastest individual service in synchronization and cloning.

We plan a number of extensions to our system as future work. We are working on techniques to automate the proof of correctness of the pPaxos algorithm. We would like to support resilience against Byzantine behavior of backend services; because clients are trusted in our model, we believe that this can be done efficiently at the cost of increasing the replication factor among participating services. Another feature is to use more efficient storage replication techniques, such as RAID, to reduce replication overhead. Most services automatically compress related objects stored by the same user; we would like to do something similar. Finally, we would like to be able to integrate other sequencing services, such as Twitter, as a pPaxos acceptor.

References

[1] 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.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] J. Constine. Dropbox hits 200m users, unveils new "for business" client combining work and personal files. http://techcrunch.com/2013/11/13/dropbox-hits-200-million-users-and-announces-new-products-for-businesses, Nov. 2013.

[9] 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.

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

[11] 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.

[12] 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.

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

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

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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, Archi-*

tectures, and Protocols for Computer Communications (SIGCOMM), pages 149–160, 2001.

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

[29] 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.

[30] 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.

[31] Y. Zhang, C. Dragga, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Viewbox: integrating local file systems with cloud storage services. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, pages 119–132. USENIX, 2014.