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
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, 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 adding, removing, or resizing a provider.

Our evaluation shows that MetaSync provides low update latency and high update throughput, close to the performance of commercial services, but is more reliable 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 200M users in November 2013, doubling its customer base over the previous year [10]. Many competing providers offer similar services, including Google Drive, Microsoft OneDrive, Box, and Baidu in China. 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 [4] or bugs [12, 20, 27], leak users' personal data [11, 28], and alter user files by adding metadata [5]. They may block access to content (e.g., DMCA takedowns [35]). From time to time, entire cloud services may go out of business (e.g., Ubuntu One [7]).

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 only. More generally, Depot [24] and SUNDR [23] showed 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. 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 user files between personal devices and the cloud. By combining multiple providers, MetaSync provides users larger storage capacity, but more importantly a higher reliability 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 (see §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 (see §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 5 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 up-

dates are reflected to all connected clients; conflicting updates are flagged for manual resolution. This usage model is supported by a background synchronization daemon (MetaSyncd shown 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 atomically using command line scripts. The system accepts an update only if it has been merged with the latest version pushed by any client. Any client can pull the latest version.

In terms of security, our baseline design allows for the backend services to be curious, unreachable, or 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 backend services have faulty implementations of their APIs or are actively malicious (§3.6). Finally, we assume that clients sharing directories and running MetaSync are trusted.

With this threat model, the goals of MetaSync are:

- **No direct client-client communication:** All clients should be able to coordinate synchronization without any direct communication among them. 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 providing 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 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 our system (§3.7). MetaSync consists of

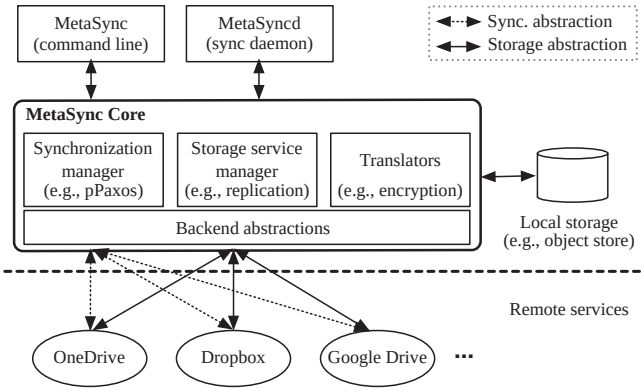


Figure 1: Overview of design. MetaSync has three main components: storage service manager to coordinate replication; synchronization manager to orchestrate cloud services; and translators to support data encryption and integrity. The components are implemented on top of an abstract cloud storage API,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

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 tear-down or 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 [18] 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 [26], where the root directory's hash is the root of the tree. This root hash uniquely defines a snapshot. Unlike git, MetaSync divides and stores each file into chunks, called Blob objects, in order to maintain and synchronize small or 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 2). A File object contains hash values of Blob objects and their offsets. A Dir object contains hash values of File objects and their names.

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

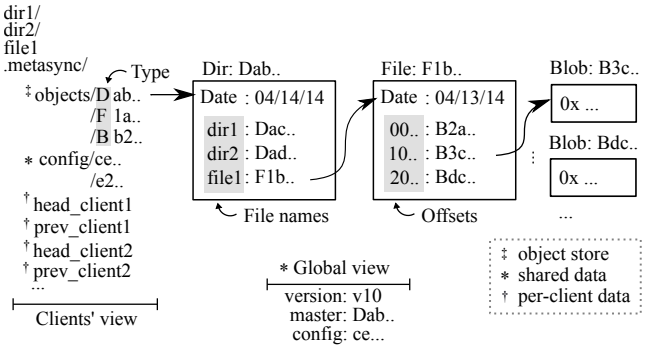


Figure 2: File management in a client's local directory. The object store maintains user files and directories with content-based addressing, in which the name of each object is based on the hash of its content. MetaSync also maintains two kinds of metadata: shared, which all clients update; and per-client, for which the owner client is the only writer. Therefore, while the object store and per-client 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).

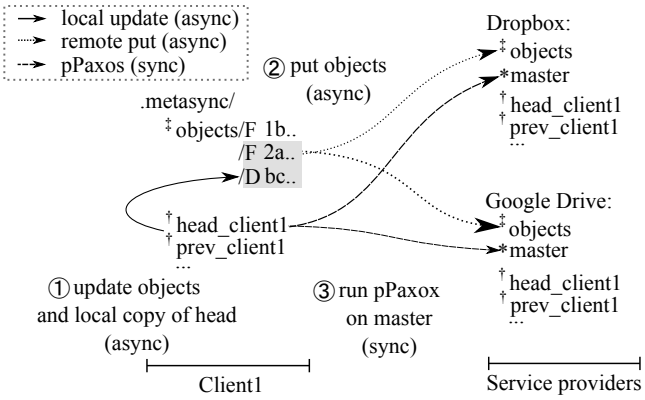


Figure 3: Workflow for check-in in MetaSync. ① MetaSync converts the file to an object, and updates its local copy of head to point to the newly-updated root directory (§3.1). ② Then, MetaSync asynchronously puts new objects redundantly into backend services, based on our mapping scheme (§3.4). ③ Finally, MetaSync runs pPaxos (§3.3) 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).

Shared data. MetaSync maintains a piece of shared data, called `master`, which is the hash value 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 data is the configuration of the backend services including information regarding the list of backends, their capacities, and authenticators. When updating any of the shared data, we invoke a synchronization protocol built from the APIs provided by existing cloud storage providers (as described in §3.3).

Per-client data. MetaSync keeps track of clients' states by maintaining a view of each client's status. The per-client data include 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. Further each client stores the copy of its per-client data into all backends after updating it.

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 based on user configuration.

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 stores the same set of files as clients, along with a subset of objects based on the mapping we explain later. A user can have multiple directories having different configurations and composition of backends to synchronize only necessary files for each client.

When files in the system are changed, an update happens as follows (see Figure 3): (1) the client updates the local objects and `head_client` to point to the current root; (2) stores the updated data blocks on the appropriate backend services; and (3) proposes its `head_client` value as the new value for `master` using pPaxos (as described in the next subsection). 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 information).

Challenges. In a distributed environment where multiple clients and storage providers are involved, it is not straightforward to coordinate updates to data that can be written by multiple clients at the same time. Since the backend storage services cannot communicate with each other in MetaSync, protocols for synchronization should be performed by clients, but without any direct communication among them (e.g., one client might be offline). Rather, clients communicate indirectly through storage providers. Furthermore, our protocol should enable each client to make progress even when some services are down or slow.

pPaxos. Creating a consistent view for the value of shared data can be reduced to the problem of having the clients come to a consensus on what is the next updated value of the data. Since clients do not have communication channels between each other, they need to rely on storage services to achieve this consensus. While the backend services do not provide support for the consensus algorithm, we devise a variant of Paxos [22], called pPaxos (passive Paxos) by using their exposed APIs. Using pPaxos, a client proposes an updated value of the shared data.

We overview pPaxos by relating it to the classic Paxos algorithm (see Figure 4(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 4(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. 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.

With a log, all clients see the 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 4. Note that the set of pPaxos acceptors need not be the same as the set of storage service backends.

Algorithm. With the availability of an append-only list abstraction, 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. Associated with each backend service is a structure that keeps the internal state of its corresponding acceptor (Figure 5 Lines 1-4). As Figure 5 shows, (1) when a client wants to propose a value, it sends a `PREPARE` to all the acceptors with a round number (Lines 7-8). This round number should be unique over proposers (e.g., client IDs could be used to break ties). Then, the client updates its status values for the acceptors by fetching and processing the acceptor logs (Lines 25-29). 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 10). (2) If there are any accepted proposals, it proposes the accepted value with the largest proposal number to the acceptors; otherwise, it proposes its own proposal with the current round number by sending `ACCEPT_REQ` to all acceptors (Lines 12-19). The `ACCEPT_REQ` will be accepted when a majority of acceptors have a smaller or equal round number (Lines 20-21, 30-32). (3) When it is accepted by the quorum, it can conclude that it has committed the new updated value (Line 23). In case it fails, it does random exponential back-off and tries again with an increased round number (Lines 33-36).

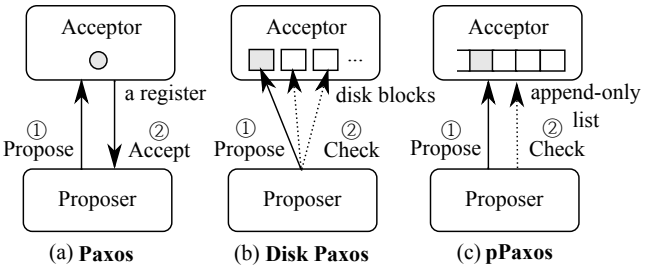


Figure 4: Comparison of operations between a proposer and an acceptor in Paxos [22], Disk Paxos [17], and pPaxos. When proposed, an acceptor in Paxos makes a decision and sends it to the proposer, whereas proposed data is stored in per-client disk blocks in Disk Paxos and in an append-only list in pPaxos. 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.

Note that this setting and the following algorithm is similar to that of Disk Paxos [17]. pPaxos can be considered as an optimized version of Disk Paxos ((Figure 4(b)). As illustrated in Figure 4, Disk Paxos requires a check of each disk block per client to determine whether the proposal is accepted, and given the passive storage interface, checking each block requires one round trip to the backend. pPaxos requires fewer round trips by taking advantage of the append-only list abstraction. Moreover, pPaxos could reach a consensus with fewer rounds than Disk Paxos because of knowing the order of messages. For Disk Paxos, once a proposer sees any proposal with a larger round number, it needs to abort its current round. A proposal that could have committed in a certain round in classic Paxos or pPaxos fails to commit in Disk Paxos merely because the proposer reads

```
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:    WaitExponentially
36:    ProposeRound(value, round, acceptors)
```

Passive Acceptor

```
37: procedure OnNewMessage(⟨msg, round⟩)
38:    Append(⟨msg, round⟩ → log)
```

Figure 5: pPaxos Algorithm.

a next-round prepare message with a larger round number; this scenario is avoided in pPaxos.

pPaxos in action. MetaSync maintains two types of shared data: `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 a single value. MetaSync utilizes a single pPaxos instance to determine and store the next value of the tuple. Then, we build functions listed in Table 1 by using multiple pPaxos instances. To do so, we maintain a pPaxos instance per synchronized value. Each client keeps the last value with which it synchronized (`prev_cilent`). When it proposes a new value, it runs pPaxos for the previous value to update it to the new value. If the instance has already accepted another value, then the client learns it as its proposal is not accepted. In this case, the client needs to find the most updated value by searching for the highest pPaxos instance number. Then, it can try to update the most updated 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 where the next value is undetermined can be considered as the most up-to-date value. Figure 6 illustrates an example snapshot of pPaxos status.

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. To the same `prev`, it always return the same value.
get_recent()	Retrieve the most recent value. It may return a stale value.

Table 1: Abstractions for consistent update.

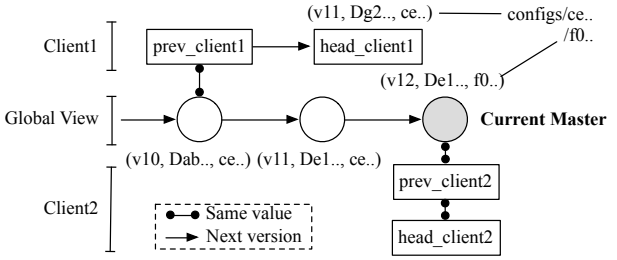


Figure 6: An example snapshot of pPaxos status with two clients. Each circle indicates a pPaxos instance. Client1 synchronized against v10. It modified some files but the changes has not been synchronized yet (head_client1). Client2 changed some files and the changes were made into v11, then made some changes in configuration and synchronized it (v12). After then, it hasn't made any changes. If client1 tries to propose the next value of v10 later, it fails. It needs to merge with v12 and creates v13 head. In addition, the client can learn configuration changes when getting v12.

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 the changes between `prev` and the current `master` into its head. To do this, MetaSync employs three-way merging as in other version control systems. It al-

lows 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 such cases, 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 Replication: Stable Deterministic Mapping

MetaSync replicates objects (in the object store) redundantly across R storage providers (R is configurable, 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, 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 storage space they provide, ranging from 2 GB in Dropbox to 2 TB in Baidu. More importantly, MetaSync's mapping scheme should consider a frequent reconfiguration of storage services (e.g., increasing storage capacity); upon changes, the re-balancing of distributed objects is guaranteed to be minimal. In this section, we describe our stable deterministic mapping algorithm, its goals, and a concrete example.

Goals. We desire a stable, deterministic mapping scheme that locates 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:

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

where H is the hash space, S is the set of services, and R is the number of replicas.

The mapping scheme 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 [21, 33]. Since it deterministically locates each object onto an identifier circle in the consistent hashing scheme, MetaSync can minimize global sharing of information among storage providers (R2).

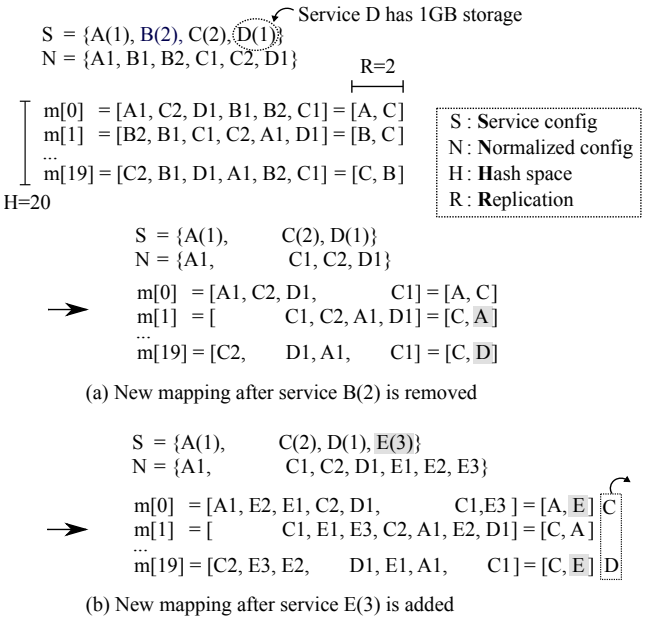
```
1:  procedure INIT(Services, HashSpace)
2:      H ← HashSpace
3:                          ▷ H: bigger values produce better mappings
4:      N ← {(sId, vId) : sId ∈ Services, 0 ≤ vId < Cap(sId)}
5:                          ▷ Cap: normalized capacity of the service
6:      for all i < H do
7:          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 7: The deterministic mapping algorithm.

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 MetaSync 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 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 (e.g., `md5` of a tuple, a service and its vnode id), 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 7. 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 [13].) 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 [34]. These are then the storage services over which MetaSync replicates the object.

Note that this mapping function takes as input the set of storage providers, the capacity settings, value of H, and a



(a) New mapping after service B(2) is removed

(b) New mapping after service E(3) is added

Figure 8: An example of deterministic mapping and its reconfigurations. The initial mapping is deterministically generated by Figure 7, given the configuration of four services, $A(1), B(2), C(2), D(1)$ where the number in a parenthesis represents the 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 $E(3)$ is added after (a). The grayed mappings indicate the new replication upon reconfiguration, and the dotted rectangle in (b) represents replications that will be garbage collected.

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 8) 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 8 illustrates 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 as in Figure 8. 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.

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 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.

Encryption translator is currently implemented using a symmetric key encryption (AES). 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 be designed to tolerate faults. MetaSync achieves fault-tolerance via replication (§3.4) for data and via pPaxos for consistency control (§3.3).

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 [18], 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

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`)	Fetch a log from `path`

Table 2: Abstractions for backend storage services.

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`.

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 [14, 19]; 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 2(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 4.

Component	Lines of code
Synchronization Manager	325
Storage service	5,099
Translators	78
Mapping scheme	258
Etc	2,339
Total	8,099

Table 3: Components of our MetaSync prototype, and their estimated complexity, in terms of lines of Python code.

3.8 Other Issues

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.

Collapsing directory All storage services manage individual files for uploading and downloading. As we see later in Table 6, 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.

4. Implementation

We have implemented a prototype of MetaSync in Python, components of which are summarized in Table 3. 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 ser-

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. 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, OneDrive: periodically list `pPaxos` directory to see if any changes since the last fetch.)
Baidu	Create a `path` directory, and consider each file as a log entry containing `msg`. For each log entry, we create a file with a monotonically 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. But it monitors all files in the account rather than a specific directory.
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 4: 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).

vice provider in Table 4. 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 4. 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`, `pull`).

5. Evaluation

This section answers the following evaluation 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.

Service	Free space (GB)	Cost ($/GB/year)
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 5: Amount of free space provided by each service and costs for additional space, as of May 2014.

Before evaluating MetaSync, we first summarize free space available from each service in Table 5. In our experiments, the free space is up to 2082 GB combining all five service providers. Users can also add more free or commercial services, or even multiple accounts in the same service. In addition to the amount of storage, 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 6 via their APIs. 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.

5.1 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 9. 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) (see Figure 5) (there could be multiple rounds in FetchNewLog de-

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 6: 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.

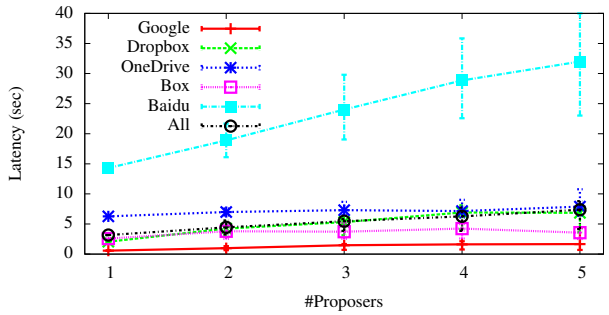


Figure 9: Latency (sec) to run a single pPaxos round with combinations of backend services and competing proposers: when using 5 different storage providers as backend nodes (all), the common path of pPaxos at a single proposer takes 3.2 sec, and the slow path with 5 competing proposers takes 7.4 sec in median.

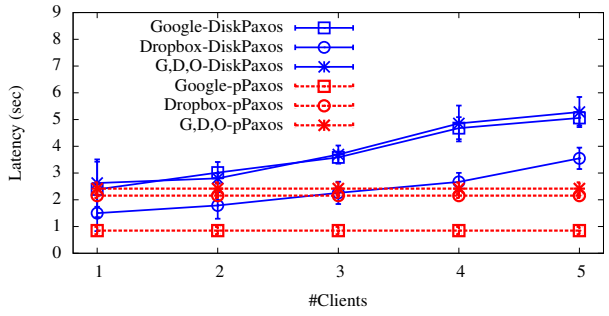


Figure 10: Comparison of latency (sec) to run a single round for Disk Paxos and pPaxos with varying number of clients when only one client proposes a value. Each line represents different backend setting; G,D,O: Google, Dropbox, and Onedrive. While pPaxos is not affected by the number of clients, Disk Paxos latency increases with it.

pending 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.

Next, we compare the latency of a single round for pPaxos with that for Disk Paxos [17]. We build Disk Paxos with service providers APIs by assigning a file as a block for each client. Figure 10 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 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 ($R = 1,2,3$). 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 7. 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%). Even for the challenging case, $S = 5, R = 3$ where an object is stored in more than a majority of services, MetaSync's mapping scheme produces distribution of objects that uses close to an even fraction of each storage, yet deterministic and resilient to reconfiguration (which we evaluate 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 the size of the calculated mapping is about 3MB.

The relocation scheme is resilient to changes as well, meaning that redistribution of objects should be minimal. As in Table 8, 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 garbage collect redundant objects from the previous configuration,

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 7: Replication results by our deterministic mapping scheme (§3.4) for Linux kernel 3.10.38 (Table 9) 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, and $R=3$ provides the strongest fault-tolerance ($f=2$), resilient against simultaneous failures of two 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 8: Time to relocate 193 MB amount of objects (photo-sharing workloads in Table 9) 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.

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.

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 9 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.

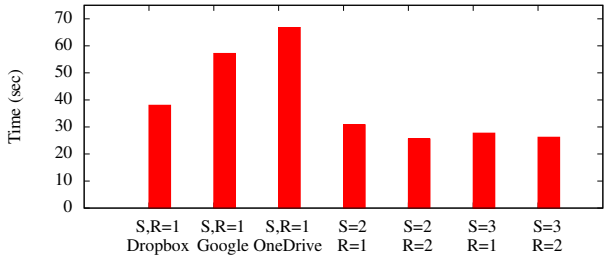


Figure 11: Time (sec) to clone an entire storage of 193 MB photos. When using individual services as a backend (Dropbox, Google, and OneDrive), MetaSync took 40-70 sec to clone, but MetaSync could improve the performance of cloning, 25-30 sec (30%) by leveraging the distributions of objects across multiple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

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 11. 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.

6. Related Work

A major line of related work, starting with Farsite [2] and SUNDR [23] but carrying through SPORC [15], Frientegrity [16], and Depot [24], is how to provide tamper resistance and privacy on untrusted storage server nodes. Un-

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 9: 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

like MetaSync, 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 [29], for example, is designed as an API for applications; thus, they delegate design choices such as how to manage files and replicate to application policy. Further, unlike MetaSync, Syndicate assumes a separate metadata service. 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 [36] 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.

Perhaps closest to our intent is DepSky [3]; it proposes a cloud of clouds for secure, byzantine-resilient storage, and it does not require code execution on the servers. The most significant difference is that we note that we can construct an atomic append primitive on all existing file synchronization services; this allows us to build our system around pPaxos, an efficient Paxos implementation for this context. DepSky, by contrast, assumes loosely synchronized clocks to support multiple writers. It 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 [22] and Disk Paxos [17]; 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 [18]; the Ori file system [25] 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 [6, 9, 23, 25, 32]. 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 [8, 30, 31], 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.

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 [37] 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, 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 available for download and use; please contact the program chair for an anonymous copy.

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] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

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

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

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

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] J. Nelson and L. Peterson. Syndicate: Democratizing cloud storage and caching through service composition. In *Proceed-*

ings of the 4th Annual Symposium on Cloud Computing, pages 46:1–46:2, 2013.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.