MetaSync: Coordinating Storage across Multiple File Synchronization Services

Cloud-based file synchronization services are a worldwide resource for many millions of users. However, individual services often have tight resource limits, suffer from outages or shutdowns, and sometimes silently corrupt or leak user data. As a solution, the authors design, implement, and evaluate MetaSync, a secure and reliable file synchronization service using multiple cloud synchronization services as untrusted storage providers. To provide global consistency among the storage providers, the authors devise a novel variant of Paxos that enables efficient updates atop unmodified APIs exported by each service. MetaSync provides better availability and performance, stronger confidentiality and integrity, and larger storage.

Seungyeop Han and Haichen Shen
University of Washington

Taesoo Kim
Georgia Institute of Technology

Arvind Krishnamurthy, Thomas Anderson, and David Wetherall
University of Washington

Cloud-based file synchronization services have become tremendously popular. Dropbox reached 400 million users in June 2015, and many competing providers offer similar services. Not only that, the increasing diversity of user devices makes these synchronization services more convenient than ever before.

Unfortunately, not all services are trustworthy or reliable: storage services routinely lose data due to internal faults[1] or bugs,[2] leak users' data,[3] and sometimes go completely out of business (see https://one.ubuntu.com/services/shutdown).

Our system (MetaSync) is based on the premise that users want the file synchronization and storage that existing cloud providers offer, but without the exposure to fragile, unreliable, or insecure services. In fact, there's no fundamental need for users to trust cloud providers, and given the aforementioned incidents, our position is that users are best served by not trusting them. Clearly, a user can encrypt files before storing them in the cloud for confidentiality. More generally, Depot[4] and Secure Untrusted Data Repository (SUNDR)[5] showed how to design systems from scratch in which users of cloud storage obtain data confidentiality, integrity, and availability without trusting the underlying storage provider. (For other work in this area, see the related sidebar.) However, these designs rely on fundamental changes to both the client and server; the focus of our research is

 Published by the IEEE Computer Society 1089-7801/16/$33.00

Related Work in Cloud File Synchronization

A major line of related work, starting with Farsite[1] and Secure Untrusted Data Repository (SUNDR)[2] and carrying through Sporc[3] and Depot,[4] is how to provide tamper resistance and privacy on untrusted storage server nodes. These systems assume the ability to specify the client–server protocol, and therefore can't run on unmodified cloud storage services.

Other systems exist that compose a storage layer on top of existing storage systems, and perhaps the closest to our intent is DepSky;[5] which proposes a cloud of clouds for secure, byzantine-resilient storage, and doesn't require code execution on the servers. Their basic algorithm assumes at most one concurrent writer. When writers are at the same local network, concurrent writes are coordinated by an external synchronization. Otherwise, it has a possible extension that can support multiple concurrent updates without an external service, which requires clock synchronization between clients.

Our implementation builds on the ideas of many earlier systems. Obviously, we're indebted to earlier work on Paxos and Disk Paxos. We maintain file objects in a manner similar to a distributed version control system like git. Content-based addressing has been used in many file systems. MetaSync uses content-based addressing for a unique purpose, letting us asynchronously upload or download objects to backend services. While algorithms for distributing or replicating objects have also been proposed and explored by previous systems, 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.

References

1. A. Adya et al., "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment," *Proc. 5th Symp. Operating Systems Design and Implementation*, 2002, pp. 1–14.
2. J. Li et al., "Secure Untrusted Data Repository," *Proc. 6th Conf. Symp. Operating Systems Design and Implementation*, 2004, pp. 1–9.
3. A.J. Feldman et al., "SPORC: Group Collaboration Using Untrusted Cloud Resources," *Proc. 9th Usenix Conf. Operating Systems Design and Implementation*, 2010; www.usenix.org/legacy/event/osdi10/tech/full_papers/Feldman.pdf.
4. P. Mahajan et al., "Depot: Cloud Storage with Minimal Trust," *Proc. 9th Usenix Conf. Operating Systems Design and Implementation*, 2010, pp. 1–12.
5. A. Bessani et al., "DepSky: Dependable and Secure Storage in a Cloud-of-Clouds," *Proc. ACM European Conf. Computer Systems*, 2011, pp. 31–46.

whether we could use existing services for these same ends.

Instead of starting from scratch, MetaSync uses multiple existing storage providers to implement a file synchronization service. We thus leverage resources that are mostly well-provisioned, normally reliable, and fairly inexpensive. While each service provides unique features, their common purpose is to synchronize a set of files between personal devices and the cloud. By combining multiple providers, MetaSync provides users with a larger storage capacity with higher availability and better performance.

The key challenge is to maintain a globally consistent view of the synchronized files across multiple clients without modifying any backend, existing services; without having a central server; and without direct client–client communication. To this end, we devise two novel methods: first, passive Paxos (pPaxos) — an efficient client-based consensus algorithm that maintains a globally consistent state among multiple passive storage backends; and second, a stable deterministic replication algorithm that supports reconfiguration (such as an adding and removing capacity) with minimal reshuffling of replicated objects.

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.

Goals and Assumptions

MetaSync's usage model 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, if desired, shares each directory with other users. Users can connect these directories with multiple devices. (We refer to the devices and software running on them as *clients* in this article.) Local updates are updated to all connected clients if users choose to use a background synchronization daemon. 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.

Our baseline design allows for backend services to be unreliable and sometimes unreachable. The storage services might try to discover which files are stored along with their content and might even accidentally corrupt or delete

files. From time to time, some services might be unavailable due to network or system failures. However, we assume that service failures are independent, services implement their own APIs correctly (except for possibly losing and corrupting user data), and communications between client and server machines are protected. This baseline model can be extended to handle faulty implementations of service APIs or actively malicious services.[6] Finally, we assume that clients sharing a specific directory are trusted, similar to a shared Dropbox directory today.

With this threat model, the goals of MetaSync are

- higher availability and performance,
- stronger confidentiality and integrity,
- greater capacity,
- no direct client–client communication,
- no additional server, and
- efficient reconfiguration.

Now that we have a grounded understanding of the system's goals, let's look at its design.

System Design

MetaSync is a distributed synchronization system that provides a reliable, globally consistent storage abstraction to multiple clients by using untrusted cloud storage services. The core library defines a set of generic APIs; all components are implemented on top of this abstraction. This makes it easy to incorporate a new storage service into our system.

Data Management

MetaSync has a similar underlying data structure to that of git[7] in managing files and their versions: objects, as units of data storage, are identified by the hash of their content. Directories form hash trees, similar to Merkle trees,[8] where the root directory's hash is the root of the tree. This root hash uniquely defines a snapshot. MetaSync divides and stores each file into chunks (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. A File object contains hash values and offsets of Blob objects. A Dir object contains hash values and names of File and Dir objects.

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

Shared metadata. MetaSync maintains a piece of shared metadata, called `master`, which is the hash value of the root directory in the most advanced snapshot. It represents a consistent view of the global state; every client needs to synchronize its status against the master. Another shared piece of metadata is the configuration of the backend services, including information regarding the list of backends, their capacities, and authenticators. To update shared metadata, MetaSync uses a special-purpose synchronization protocol built from the APIs provided by existing cloud storage providers.

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

Overview

MetaSync's core library maintains the aforementioned data structures and exposes a reliable storage abstraction to the users. The library's role is to mediate accesses and updates to actual files and metadata, and further interact with the backend storage services to make file data persistent and reliable.

Initially, a user sets up a directory to be managed by MetaSync; files and directories under that directory will be synchronized. Each managed directory has a name (called namespace) in the system to be used in synchronizing with other clients. Upon initiation, MetaSync creates a folder with the name in each backend. The folder at the backend storage service stores the configuration information plus a subset of objects. A user can have multiple directories

with different configurations and a composition of backends.

When a client changes a file (or set of files), an update happens: the first step is that the client updates the objects in the object store and `head_client` to point to the new root Dir object; then in step 2, the library stores the updated objects on the appropriate backend services; and in step 3, the system proposes its `head_client` value as the new value for master using pPaxos. Steps 1 and 2 don't require any coordination, as step 1 happens locally and step 2 proceeds asynchronously. Crucially, a client doesn't have to update global `master` for every local file write. The `head_client` and `prev_client` are updated to all the clients before step 2, and objects that aren't referenced by any of them or their descendants in the hash trees are later removed by clients through periodic checking.

Consistent Update of Global View: pPaxos

The file structure we described allows MetaSync to minimize the use of synchronization operations. Each object in the object store can be independently uploaded, because it uses content-based addressing. Each per-client data file (such as `head_client`) is also independent, because 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 (for example, `master` and configuration).

In a distributed environment, it's not straightforward to coordinate updates to data that can be modified by multiple clients simultaneously. To create a consistent view, clients must agree on the sequence of updates applied to the shared state.

Clients don't have communication channels between each other (such as they might be offline), so they need to rely on the storage services to achieve this consensus. However, these services don't communicate with each other, nor do they implement consensus primitives. Instead, pPaxos (our variant of Paxos)[9] uses the exposed APIs of these services.

Paxos is a multiround, non-blocking consensus algorithm that's safe regardless of failures, and makes progress as long as a majority is alive. Paxos would be sufficient for coordinating updates, except that the backend services don't implement this interface. Instead, we only rely on them to provide an *append-only* list that *atomically* appends an incoming message at the end of the list. This abstraction is either readily available or can be layered on top of the API provided by existing storage services. For example, we use Google Drive comments and Dropbox versioning.

With this append-only list abstraction, we can simulate Paxos. The backend services act as *passive acceptors*. Clients determine which proposal was accepted by examining the message log to see what the service would have done if it had been running Paxos. Each client proposes the new shared value as the next operation, and it's accepted if the majority of backend services agree on it after reading the logs. A detailed description of the algorithm is available elsewhere.[6]

This setting is similar to the motivation behind Disk Paxos,[10] an earlier algorithm that implements Paxos across passive network-attached storage devices; indeed, pPaxos can be considered as an optimized version of Disk Paxos. Disk Paxos assumes that each storage device provides only a simple block interface. Clients write proposals to their own dedicated block on each disk, but they must check everyone else's blocks to determine the outcome. Thus, the number of messages in Disk Paxos for a single proposal is proportional to the product of the number of servers and clients; the number of messages in pPaxos is only proportional to the number of servers. Figure 1 highlights the differences among pPaxos, Disk Paxos, and original Paxos algorithms.

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

The version is a monotonically increasing number that's uniquely determined for each `master_hash`, `config_hash` pair. This tuple is used in pPaxos to describe a client's status, and is stored in `head_client` and `prev_client`.

The pPaxos algorithm can determine and store the next value of the three-tuple. Each client keeps the last value with which it synchronized (`prev_client`). To propose a new value, the client runs pPaxos to update the previous value with the new value. If another value has already been accepted, the client can try to update the new value after they merge. 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 to the next hash value, and the tail of the list is the most up-to-date.

Note that merging is required when a client synchronizes its local changes (`head`) with the current master that's different from what the client previously synchronized (`prev`). In this case, proposing the current head as the next update to `prev` via pPaxos returns a different value than the proposed head — the proposal fails. Some of the conflicts can be automatically resolved through three-way merging. Otherwise, MetaSync just marks the conflict so that users can resolve it manually.

Replication: Stable Deterministic Mapping

MetaSync replicates objects (in the object store) redundantly across R storage providers (R is configurable; typically $R = 2$) to provide high availability even when a service is temporarily inaccessible. This also provides potentially better performance over wide-area networks. Because R is less than the number of services, we must maintain information regarding the mapping of objects to services. MetaSync requires a mapping scheme that takes into account storage space limits imposed by each storage service; if handled poorly, lack of storage at a single service can block the entire operation of MetaSync, and typical storage services vary in the (free) space they provide, ranging from 2 Gbytes in Dropbox to 2 Tbytes in Baidu. In addition, the mapping scheme should consider a potential reconfiguration of storage services (such as increasing storage capacity); upon changes, the rebalancing of distributed objects should be minimal. In our setting, the mapping should meet three requirements (R):

- R1 — support variations in storage size limits across different services and across different users;
- R2 — share minimal information among services; and
- R3 — minimize the realignment of objects upon removing or adding a service.

Instead of maintaining the mapping information of each object, we use a stable, deterministic mapping function that locates each object to a group of services over which it's replicated; each client can calculate the same result independently given the same object. Given a hash of an object (mod H), the mapping is map: $H \rightarrow \{s : |s| = R, s \subset S\}$, where H is the hash space, S is the set of services, and R is the number of replicas. To provide a balanced mapping that takes into account storage variations across services (R1), we could use a mapping scheme that represents services with different storage capacities as a variable number of virtual nodes in a consistent hashing algorithm.[11,12] Because the consistent hashing scheme deterministically locates each object on an identifier circle, MetaSync can minimize information shared among storage providers (R2).

However, using consistent hashing has two problems. First, an object can be mapped to multiple virtual nodes corresponding to the same service, reducing availability. Second, a change in a service's storage capacity means changing the number of virtual nodes; if this changes the size of the hash space, many or most objects will need to be shuffled (R3). To solve these problems, we introduce a stable, deterministic mapping scheme that maps an object to a unique set of virtual nodes and minimizes reshuffling when resource availability changes. The key idea is to achieve the random distribution via hashing and stability of remapping by sorting these hashed values.

The stable deterministic mapping scheme uses multiple virtual storage nodes for each provider, where the number of virtual nodes is proportional to the capacity of that provider for a given user. Then MetaSync divides the hash space into H partitions. H is configurable, but remains fixed even as the service configuration changes; larger values produce better-balanced mappings for heterogeneous storage limits. During initialization, MetaSync assigns each of the H partitions a different, ordered list of virtual nodes. The ordering depends on the hash of the partition index, the service ID, and the virtual node ID. Given an object hash n, the data are stored on the first R distinct services from the list associated with the (n mod H)th partition.

The mapping function takes as input the set of storage providers, capacity settings, value of H, and a hash function. The virtual node list is populated proportionally to service capacity, and the ordering in each list is determined by a uniform hash function. Thus, the resulting mapping of objects onto services should be

proportional to service capacity limits with H. When a virtual node is added or removed (synchronized through an update to shared metadata), the amount of data that must be shifted is proportional to the virtual node's size.

Figure 2 shows an example of our mapping scheme with four services providing 1 or 2 Gbytes of free spaces — for example, A(1) means that service A provides 1 Gbyte of free space. Given the replication requirement (R = 2) and the hash space (H = 20), we can populate the initial mapping. Figures 2a and 2b illustrate the realignment of objects upon the removal of service B(2), and the inclusion of a new service E(3), respectively.

When reconfiguration happens, the client first uploads all the newly added objects to backends, then modifies its configuration file and updates the shared data with the new `config_hash` through pPaxos. Finally, it removes unreferenced objects from backends.

Translators

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

The encryption translator is currently implemented using a symmetric key encryption (AES-CBC). MetaSync keeps the encryption key locally and doesn't store it on the backend. When a user clones the directory in another device, the user must provide the encryption key. An integrity checker runs a hash function over a retrieved object and compares the digest against the file name. If it doesn't match, it drops the object and downloads the object by using other backends from the mapping. This must be performed only as part of the `get` chain.

Evaluation

To evaluate the system's design and usefulness, here we answer the following questions: What are the performance characteristics of pPaxos? And what's the end-to-end performance of MetaSync?

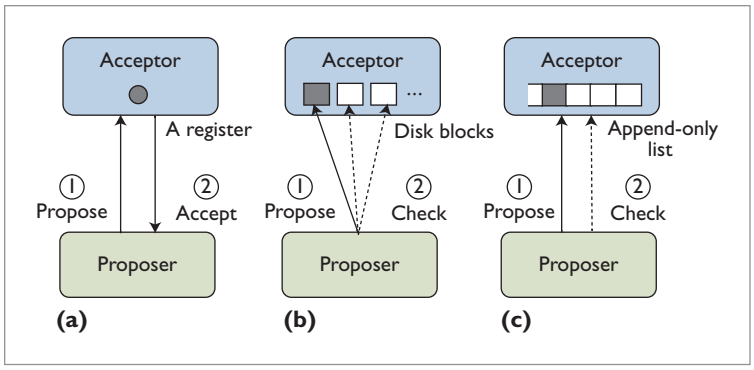

Figure 1. Comparison of operations between a proposer and an acceptor in (a) Paxos, (b) Disk Paxos, and (c) passive Paxos (pPaxos). Each acceptor in Paxos makes a local decision to accept or reject a proposal and then replies with the result. Disk Paxos assumes acceptors are passive; clients write proposals into per-client disk blocks at each acceptor. Proposers need to check every per-client block (at every acceptor) to determine if their proposal was accepted, or pre-empted by another concurrent proposal. With pPaxos, the append-only log lets clients efficiently check the outcome at the passive acceptor.

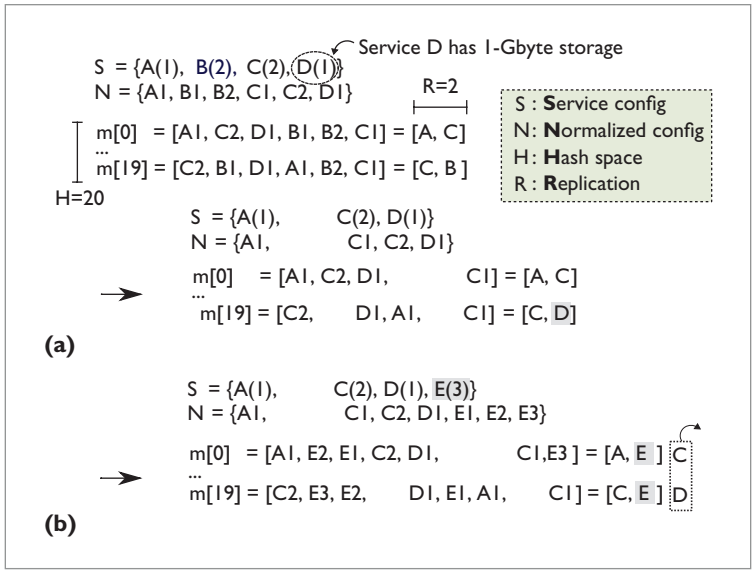

Figure 2. An example of deterministic mapping and its reconfigurations. (a) New mapping after service B(2) is removed. (b) New mapping after service E(3) is added. The grayed mappings indicate the new replication upon reconfiguration, and the rectangle in (b) represents replications that will be removed.

Performance of pPaxos

We measure how quickly pPaxos reaches consensus as we vary the number of concurrent proposers. Figure 3 shows the results of the experiment with 1–5 proposers over five storage providers. A single run of pPaxos took about 3.2 seconds on average under a single writer model to verify

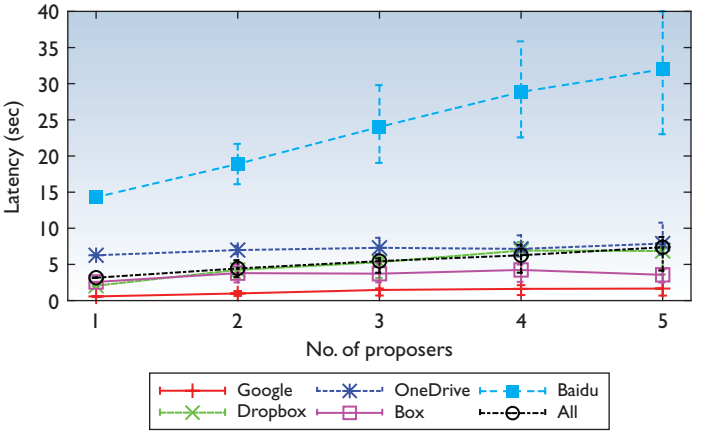

Figure 3. Latency (in seconds) to run a single pPaxos round with combinations of backend services and competing proposers. Each measurement is done five times, and each shows the average latency.

acceptance of the proposal when using all five storage providers. This requires at least four roundtrips: `PREPARE` (Send, FetchNewLog) and `ACCEPT` (Send, FetchNewLog); there could be multiple rounds, depending on the implementation for each service. It took about 7.4 second with five 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.

End-to-End Performance

We selected three workloads to demonstrate performance characteristics. First, the Linux kernel source tree represents the most challenging workload for all storage services, due to its large volume of files and directory (920 directories and 15,000 files, with a total of 166 Mbytes). Second, MetaSync's paper workload represents a causal use of synchronization service for users (three directories and 70 files, with a total of 1.6 Mbytes). Third, sharing photos is for maximizing the throughput of each storage service with bigger files (50 files, with a total of 193 Mbytes).

Table 1 summarizes our results for end-to-end performance for all workloads, comparing MetaSync with the native clients provided by each service. For $S = 5$, $R = 1$, using all five services without replication, MetaSync provides comparable performance to native clients at median speed for MetaSync paper and photo sharing, but outperforming for Linux kernel workloads. However, for $S = 5$, $R = 2$, which involves replicating objects twice, MetaSync is more than 10× faster than Dropbox in a Linux kernel and 2.3× faster in photo sharing.

We should note that each workload was copied into one client's directory before synchronization began. 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. MetaSync outperforms any individual service for all workloads. Especially for the Linux kernel source, it took only 12 minutes when using four services (excluding the Baidu located outside of the country), compared to more than 2 hours with native clients. This improvement is possible due to using concurrent connections to multiple backends, as well as optimizations such as collapsing directories. Although these native clients might not be optimized for the highest possible throughput,

Table 1. Synchronization performance of five native clients provided by each storage service, and with four different settings of MetaSync.

						MetaSync			
Workload	**Dropbox**	**Google**	**Box**	**OneDrive**	**Baidu**	**$S = 5, R = 1$**	**$S = 5, R = 2$**	**$S = 4, R = 1$**	**$S = 4, R = 2$**
Linux kernel source	2 h 45 m	>3 hrs	>3 hrs	2 h 3 m	> 3 hrs	1 h 8 m	13 m 51 s	18 m 57 s	12 m 18 s
MetaSync's paper (seconds)	48	42	148	54	143	55	50	27	26
Photo sharing (seconds)	415	143	536	1,131	1,837	1,185	180	137	112

considering that they can run as a background service, it would be beneficial for users to have a faster option.

MetaSync provides a secure, reliable, and performant file synchronization service on top of popular cloud storage providers. It supports five commercial storage backends (in the current open source version), and outperforms the fastest individual service in synchronization and cloning, by 1.2 to 10× on our benchmarks.

MetaSync is publicly available for download and use (see http://uwnetworkslab.github.io/metasync/). In future work, we'll extend our system to work with byzantine fault services and other use cases, such as mobile applications.

Acknowledgments

This work was supported by the US National Science Foundation (CNS-0963754, 1318396, and 1420703) and Google. This material is based on research sponsored by DARPA under agreement FA8750-12-2-0107. The US government is authorized to reproduce and distribute reprints for governmental purposes, notwithstanding any copyright notation thereon. Taesoo Kim was partly supported by MSIP/IITP (B0101-15-0644).

References

1. C. Brooks, "Cloud Storage Often Results in Data Loss," *Business News Daily*, 8 Oct. 2011; www.businessnewsdaily.com/1543-cloud-data-storage-problems.html.
2. G. Huntley, "Dropbox Confirms That a Bug within Selective Sync May Have Caused Data Loss," *Y Hacker News*, Oct. 2014; https://news.ycombinator.com/item?id=8440985.
3. J. Cook, "All the Different Ways that 'iCloud' Naked Celebrity Photo Leak Might Have Happened," *Business Insider*, 1 Sept. 2014; www.businessinsider.com/icloud-naked-celebrity-photo-leak-2014-9.
4. P. Mahajan et al., "Depot: Cloud Storage with Minimal Trust," *Proc. 9th Usenix Conf. Operating Systems Design and Implementation*, 2010, pp. 1–12.
5. J. Li et al., "Secure Untrusted Data Repository," *Proc. 6th Conf. Symp. Operating Systems Design and Implementation*, 2004, pp. 1–9.
6. S. Han et al., "MetaSync: File Synchronization across Multiple Untrusted Storage Services," *Proc. 2015 Usenix Ann. Technical Conf.*, 2015; www.usenix.org/conference/atc15/technical-session/presentation/han.
7. S. Chacon and B. Straub, "Git Internals – Git Objects," *Pro Git*, APress, 2014; http://git-scm.com/book/en/Git-Internals-Git-Objects.
8. R.C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," *Proc. 7th Ann. Int'l Cryptology Conf.*, 1987, pp. 369–378.
9. L. Lamport, "The Part-Time Parliament," *ACM Trans. Computer Systems*, vol. 16, no. 2, 1998, pp. 133–169.
10. E. Gafni and L. Lamport, "Disk Paxos," *Distributed Computing*, vol. 16, no. 1, 2003, pp. 1–20.
11. D. Karger et al., "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," *Proc. 29th Ann. ACM Symp. Theory of Computing*, 1997, pp. 654–663.
12. I. Stoica et al., "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *Proc. 2001 Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm.*, 2001, pp. 149–160.

Seungyeop Han is a PhD candidate in the Computer Science and Engineering Department at the University of Washington. His research interests include networked systems, distributed systems, computer networks, and security and privacy. Han has an MS in computer science and engineering from the University of Washington. Contact him at syhan@cs.washington.edu.

Haichen Shen is a PhD student in the Computer Science and Engineering Department at the University of Washington. His research interests include mobile computing, computer networking, and distributed systems. Shen has an MS in computer science and engineering from the University of Washington. Contact him at haichen@cs.washington.edu.

Taesoo Kim is an assistant professor in the School Computer Science at Georgia Institute of Technology. His research interests include designing and building secure systems from trusted components. Kim has a PhD in computer science from the Massachusetts Institute of Technology. Contact him at taesoo@gatech.edu.

Arvind Krishnamurthy is an associate professor in the Computer Science and Engineering Department at the University of Washington. His research interests include all aspects of building practical and robust computer systems, especially with regards to the robustness, security, and performance of Internet-scale systems. Krishnamurthy has a PhD in computer science and engineering from the University of California, Berkeley. Contact him at arvind@cs.washington.edu.

Thomas Anderson is the Warren Francis and Wilma Kolm Bradley Chair of Computer Science and Engineering at the University of Washington. His research interests include all aspects of building practical, robust, and efficient computer

systems, especially distributed systems, operating systems, computer networks, multiprocessors, and security. He was recently elected to the National Academy of Engineering and is an ACM Fellow. He's the winner of the Usenix Lifetime Achievement Award, the Usenix Software Tools User Group Award, the IEEE Koji Kobayashi Computer and Communications Award, the ACM Sigops Mark Weiser Award, and the IEEE Communications Society William R. Bennett Prize. Contact him at tom@cs.washington.edu.

David Wetherall is a principal engineer at Google, and is a former professor of computer science and engineering at the University of Washington. He research interests include the design of datacenter and backbone networks. Wetherall has a PhD in computer science from the Massachusetts Institute of Technology. He's a Fellow of ACM and IEEE. Contact him at djw@cs.washington.edu.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.