A social link based private storage cloud

Michalis Konstantopoulos  
University of Athens, Greece  
mkonstan@di.uoa.gr

Nikos Chondros  
University of Athens, Greece  
n.chondros@di.uoa.gr

Mema Roussopoulos  
University of Athens, Greece  
mema@di.uoa.gr

Abstract—In this paper, we present $O^3$, a social link based private storage cloud for decentralized collaboration. $O^3$ allows users to share and collaborate on collections of files (shared folders) with others, in a decentralized manner, without the need for intermediaries (such as public cloud storage servers) to intervene. Thus, users can keep their working relationships (who they work with) and what they work on private from third parties. Using benchmarks and traces from real workloads, we experimentally evaluate $O^3$ and demonstrate that the system scales linearly when synchronizing increasing numbers of concurrent users, while performing on-par with the ext4 non-version-tracking filesystem.

Index Terms—Private Storage Cloud, Distributed File Systems, Replication, File sharing, Synchronization

I. INTRODUCTION

Today, naming, addressing, and most of the Internet services that critically affect users’ lives are in the hands of a few large, centralized, and typically, corporate entities. One critical Internet service with a profound impact on users’ lives is that of data storage. A number of popular cloud storage providers exist that offer features such as storage, anytime-anywhere access, and automated file synchronization and storage maintenance (e.g., [3], [10], [13], [25]). While convenient, these services come with a critical privacy cost. To obtain service, users inevitably hand over to the service provider access to personal information including a subset (or all) of their private data collections, or/and information about their social and working relationships with other users.

There is an alarming rise in the number of incidents from major cloud storage providers involving data leakage, data loss, intentional deletion of user files, and systematic sifting and analysis of user data for commercial benefit (e.g., [6], [11], [16], [18], [23], [41]). As a result, there have been growing calls from researchers and society at large in favor of "re-decentralizing" the Internet, taking its services out of the hands of large corporate entities. This work is a step in this direction.

In this paper, we present the “On Our Own” ($O^3$) social link-based private storage cloud. $O^3$ allows users to share collections of files (shared folders) with others, in a decentralized manner, without the need for intermediaries (such as public cloud storage servers) to intervene. Thus, users can keep their relationships with other users (who they socialize and work with) and their activities (what they share and what they work on) private from third parties.

$O^3$ provides a uniform POSIX interface for legacy applications accessing both a user’s shared folders as well as her private (non-shared) folders. Users can partially replicate any subset of their data collections (shared or non-shared) on their devices of choice. Thus, $O^3$ can be used by users to holistically manage their entire cross-device data collections. To this end, the system provides a global view of all of a user’s files, regardless of whether they are stored locally or not, and can fetch files on demand from other devices owned by the user, or in the case of shared files, from other users’ devices.

We summarize the contributions of this paper as follows. We present the design of the $O^3$ private storage cloud for private, decentralized collaboration. Users can work at leisure on files while the system 1) seamlessly tracks all changes that users make to files, 2) automates the synchronization of replicated files across users and their devices, and 3) detects and helps users resolve conflicts via 3-way merge that arise from concurrent updates made when devices are offline. $O^3$ achieves this with a synchronization protocol and algorithms based on novel data structures called “epoch-graphs” that enable the system to track and propagate changes to files and enforce causality-preserving replay of these changes across all devices of all users. The protocol, algorithms, and epoch graphs are carefully designed to respect user privacy; they intentionally hide information about a user and her work habits, such as the number of devices the user owns and the user’s rate of work (e.g., updates made) on files she has not shared with others. The system provides seamless treatment of both a user’s shared as well as her private, non-shared files distributed across her personal devices. Finally, using benchmarks and traces from real workloads, we experimentally evaluate $O^3$ and demonstrate that the system scales linearly when synchronizing increasing numbers of concurrent users, while performing on-par with the ext4 non-version-tracking filesystem.

II. SYSTEM OVERVIEW

$O^3$ is a distributed, multi-user, multi-device per user, file system. We assume an asynchronous network, lossless channels, full connectivity between nodes, and focus on crash fault tolerance. $O^3$ allows users to:

1) manage, replicate, and synchronize data across their own devices,
2) share subsets of their data with other users, work independently on the shared files, and synchronize in an ad-hoc manner, i.e., work in an offline fashion.
\(O^3\) manages replication of items using Replication Rules, and assists the user in resolving the conflicts that occur when multiple users work on the same file at different devices.

\(O^3\) is a POSIX-compliant file system that allows unmodified applications to run on top of it. It tracks the versions of files as they change, upon every file-system close event, and manages these versions in version graphs. If conflicts arise, it uses these version graphs to present all “current” versions of a file to the user, using a different “decorated” name for each. The user can further edit any of these versions and select one of them as the resolution of the conflict, using an \(O^3\)-specific tool.

\(O^3\) supports partial replication of file content, by allowing the user to define Replication Rules. Each such rule addresses a subset of her data collection and declares the devices that should store the content of matching files. The system enforces these rules automatically.

\(O^3\) consists of the following major subsystems:

1. A Metadata Store (MS), which tracks all object metadata, such as name, location, timestamps, etc.
2. A Content Store (CS), which manages file content in a copy-on-write manner.
3. A Replication Rules Enforcer subsystem, that applies the rules by identifying missing content, fetching it, and storing it in the CS.
4. A Synchronization subsystem that fully replicates the MS.
5. A POSIX-compliant file system that provides the interface to \(O^3\) for user-space applications.
6. An asynchronous communications layer that integrates device management, user-to-user communications, and presence notifications, forming a Decentralized Connectivity Service (DCS) between devices of collaborating users.

In Figure 1 we present the subsystems’ interaction when accessing a file’s data, and in Figure 2 we present the subsystems’ interaction when synchronizing data and metadata.

The MS is the core component of the system, and is fully replicated on all of a user’s devices. It consists of a metadata-repository and an update-log. The former is a database of metadata for all objects stored in the system. Each object is identified by an Object Identifier (Oid), which is globally unique across all users. An object can be directory, a file, a replication rule, a snapshot of the system, and a share (see Section III-A). Each object can have multiple versions, each identified by a globally unique version id (Vid).

Versions form a directed acyclic graph (DAG) from the root version, all the way to potentially multiple versions that are leaves in the version graph. The DAG always has a single root, as an object can only be created on one device. Each version contains metadata that fully describe the object’s state as of this version. For file objects, one entry in this metadata is a Content Identifier (Cid), which points to the CS element that contains this file version’s data. Directory objects may “contain” other objects. We track this relationship in the metadata of each version of the contained object, where we store a pointer to the directory object that “hosts” it. We maintain a reverse index for every directory object, and have the MS answer queries such as “which files exist in directory with Oid”, with the leaf versions of the contained objects.

The MS repository is optimized to quickly answer queries that arrive from the file system. The MS update-log is the transaction log that tracks all update actions that happen on the MS repository, and is used to efficiently replicate the repository across nodes. Its structure and use is presented in Section III.

The CS is the central repository of all file data. It manages elements, each identified by a globally unique Content Identifier (Cid). Thus, for CS, a Cid is opened, read, written, and closed. Our design allows for multiple CS implementations, but enforces copy-on-write semantics: whenever a Cid is opened for writing, we expect the CS implementation to leave the original contents of the element with Cid unchanged.

The Replication Rules Enforcer subsystem applies the Replication Rules the user has defined. Each rule is specified as a (Directory prefix, device list) tuple. For the current device d, the enforcer identifies all rules that target d. For each rule it creates a set of Cids of versions of objects that have a location that matches the rule’s directory prefix. It then proceeds to collect from the MS all Cids missing from the current device and issues requests to fetch the content associated with them.

The file system (FS) is the layer that integrates all the above components and provides a way for applications to interact with \(O^3\). The FS presents a global view of all files stored on all the user’s devices, at each device. This is feasible because the MS is fully replicated across all nodes. Because of this, the FS may be asked to open a file whose content is missing from the current device. In this case, it interacts with the communications layer, fetches the content on-demand (if possible), and presents it to user-space.

The FS parses a file path, element by element, and looks up each path element in the MS, from which it obtains the corresponding object’s metadata, in their “current” form, as

![Diagram of O^3 components when accessing the contents of a file.](image)
Fig. 2: Communication of $O^3$ components during the execution of EGSync and Replication Rules Enforcer, between Alice at device X and Bob at device Y.

denoted by the object’s version graph. It provides this metadata to the application, and allows user-space programs to access files, directories, and symbolic links as needed. When the FS encounters object version graphs with multiple leaves (denoting files in conflict), it provides these leaf versions and the common parent (i.e., the closest common ancestor) with “decorated” names, so the user can identify the conflict quickly and resolve it accordingly via 3-way merge.

The DCS component allows users (and their applications) to establish point-to-point connections with other users, without the use of a centralized party. A user defines her list of contacts and stores them locally using aliases. DCS consists of a local Contact Management Service component that resides on the user’s local device and stores and manages user’s contacts and a set of Distributed Hash-Table (DHT) nodes. Together, they facilitate connectivity between applications by providing an abstraction for establishing peer-to-peer connections regardless of the network topology. This component is the subject of a separate article [4].

In this paper, we focus on inter-user sharing, and the MS synchronization protocol and algorithms of $O^3$. For a detailed presentation of the remaining components listed above, we refer the reader to [21].

III. SHARING AND SYNCHRONIZATION

Users create files (e.g., documents, photos) regularly, and replicate their contents to their own personal devices. Additionally, they may share some of these files with other users, who are free to obtain, modify, and synchronize them continuously. In this section, we describe how $O^3$ provides replication and sharing using the MS update-log (Section III-A) and a new synchronization protocol (Section III-B).

A. MS Update Log

We design the MS update log (See Section II) around a novel data structure called Epoch Graph (EG). When a user performs a file system update operation (e.g. edit, delete, rename), $O^3$ generates a new version in the corresponding object’s version graph. Additionally, $O^3$ registers a pointer to this version in an epoch, a structure that stores all versions generated during a specific period of time. An epoch is identified by a globally unique identifier Eid. In our current design, the Eid comprises a tuple (userID, deviceID, tag), where the userID and deviceID identify the user and device that created the epoch, and the tag uniquely identifies the epoch within this device. Each device maintains a “current” epoch to record new versions as they occur, which it closes at intervals determined locally. Finally, an epoch E has a dependency list P of other “parent” epochs; all object versions recorded at each epoch p in P are incorporated into the metadata repository, before epoch E is closed.

Figure 3 depicts a Private EG, which is the $O^3$ uses to record non-shared objects. The first epoch node in the Private EG is the root epoch, which is an empty epoch with a well known identifier; each user’s device initializes its copy of the Private EG with this node, and creates a new “current” epoch ready to record local MS repository changes (i.e. object update operations). EG grows over time as devices learn about each other’s (potentially concurrent) epochs via synchronization sessions.

![Epoch Graph Example](image)

Fig. 3: A sample epoch graph depicting concurrency between different devices of a user. Devices A and B synchronized immediately after epoch A2, and then worked in isolation until epoch A5, which was closed after device A incorporated epoch B2 created at device B.

When a device closes its current epoch E (e.g., when a synchronization event occurs), it sets the dependency list P of E to the current leaves of the Private EG, and thus E becomes the new, sole leaf of the Private EG. This creates a single rooted, directed, acyclic graph of epochs, which we store with both directions. Additionally, we store Eids in a dictionary, allowing random access to any node in the graph.

$O^3$ supports sharing of files across users at the granularity of folders. A share is a folder the user selects to collaborate on with others. The folder may have an initial set of objects in it, and grow as users collaborate and add to it over time. For every share, there is a separate, independent epoch graph. Thus, the update log includes the Private EG and a separate EG for each share in which the user participates.

Any user can create a share locally at any of her devices, selecting its participants. Then $O^3$ performs the following:

- It creates a new Share EG, along with its root epoch.
It creates a share object, which contains the Sid (a globally unique identifier, which contains (userID, deviceID, tag)), the share’s root Eid, and the list of users participating in the share. Then, it records this object in the current epoch of the Private EG, so that the remaining personal devices of the user eventually learn about the share, during subsequent synchronization sessions.

For each file system object that already exists in the folder to share, the system “obsoletes” it and creates a “shallow” copy of its leaf versions into a new object, thus guaranteeing that the pre-existing version history is not exposed to the other users participating in the share. “Obsoleting” an object is actually a deletion event, where a new version is created at its version graph with parents all current leaves; this deletion event is marked in the Private EG. The new object, which contains copies of the leaf versions of the obsoleted one (along with their common parent in case of conflict), is registered in the root epoch of the share’s EG.

The share’s creator (e.g., Alice) notifies out-of-band the participating users (e.g., Bob) with the Sid and the list of participants. With this information, Bob is able to mount the share and fetch its data from any participant’s device. During mount, O3 creates a local copy of the share definition in Bob’s Private EG and creates a new EG for the data of the share. We apply per-share access control where a user is allowed to receive or send updates over shared data, only if she is already a participant of the share. The membership is initially defined during share creation, and its owner is allowed to edit it, by adding or removing members via addUser and removeUser actions. Finally, the owner is free to transfer her ownership to another user of the share via the transferOwnership action. These actions are invoked through the O3-specific tool and create new versions of the share in MS.

Epochs are a means to synchronize the MS repository across devices, as further explained in Section III-B below. They are efficient storage-wise, as they are simply an ordered list of (oid, Vid) tuples. Nonetheless, they are often garbage collected to preserve space. For the user’s private EG, an epoch E is discarded when every device reports an epoch with a path towards E. For a share EG, an epoch E at a user U’s device is discarded when every device of U and at least one device of every other user participating in the share, reports an epoch with a path towards E.

B. Synchronization Protocol

O3 replicates the MS repository using the EG data structure and the two-phase Epoch Graph Synchronization (EGSync) protocol. An O3 device initiates an EGSync instance to obtain the epochs of another device’s EG (either the Private or a Share-specific one). Figure 4 provides a UML sequence diagram of the protocol between Alice and Bob. Note that the identifiers Sid and Eid in Figure 4 are actually tuples (userID, etag) where etag is the encrypted (deviceID, tag) described in Section III-A. The personal devices of a user share a key, which they use to encrypt private portions of each identifier they expose to other users. This choice allows the system to hide both the cardinality of a user’s personal device group (by hiding device ids), as well as counters used to produce identifiers, that would expose the progress made by the user on objects outside the share.

1) First phase – missing epochs discovery: Alice initiates a new protocol instance by sending a SessionInit message with current knowledge CK, which is the set of Eid’s of the leaves of EG_Alice, as recorded in her current device. CK is necessary for Bob to efficiently locate the epochs in EG_Bob missing from EG_Alice. Bob uses Algorithm 1 to produce PNK, which is the set of epochs potentially not known to (i.e., potentially missing from) EG_Alice.

In line 2, the algorithm detects if every epoch in the remote’s (Alice’s) current knowledge set (RCK) exists in the current (Bob’s) device’s EG (EG_Bob). If all epochs are found, it initializes a starting set SS with RCK. If however any epoch in RCK is missing, it initializes SS with the latest known epochs (line 8) created by Alice’s devices, that Bob knows about. The latest known epochs of Alice are accessible because each device in O3 maintains locally the latest known epochs for each user, which are updated when new epochs arrive during a synchronization session. Multiple latest known epochs for Alice may exist, if Alice worked concurrently across more than one of her devices.

Given SS, the algorithm locates the nearest trunk node (line 11). We define a node of an EG as trunk when it has no siblings (e.g., nodes A1,A2,A5 of Figure 3). The algorithm...
then starts descending $\text{EG}_{Bob}$ towards the leaves, adding nodes it encounters to the $\text{PNK}$ set (function $\text{descendants}$ in line 12). From this set, it subtracts all nodes encountered in all paths from $SS$ up to the nearest trunk that started the search, as at least one of Alice’s devices already knows about them (function $\text{ancestorsUntil}$, in line 12). If the $\text{RCK}$ is an empty set (line 3), then this node has no data for the share and this is its initial synchronization session; thus $\text{PNK}$ is populated with all epochs of $\text{EG}_{Bob}$ (line 5). For graph traversals, we use breadth-first search while memorizing visited nodes, to avoid repeated visits due to multi-parented nodes. These traversals are efficient, as $\text{EG}$ is maintained using both directions and the connectivity is proportional to the number of nodes between the entry and exit nodes of the traversal. This number grows when devices reduce the rate of synchronization, so a frequent rate would reduce the complexity.

**Algorithm 1** Potentially not known epochs to Remote

```
1: function $\text{PNKEpochs}(\text{EG}, \text{RCK}, \text{userID})$:  \( \triangleright \) RCK: remote CK
2: allKnown $\leftarrow \forall e \in \text{RCK}: e \in \text{EG}$
3: $SS$ $\leftarrow$ \( \forall e \in \text{EG} \) $\triangleright$ starting points for $\text{PNK}$ search
4: if $\text{RCK}$ $= \emptyset$ then
5: $\text{PNK}$ $\leftarrow$ $\text{EG}.\text{root} \cup \text{descendants}(\text{EG}.\text{root})$
6: else
7: if $\neg$allKnown then
8: $SS$ $\leftarrow$ latestKnownEpochs($\text{userID}$)
9: else
10: $SS$ $\leftarrow rCK$
11: trunk $\leftarrow$ nearestTrunk($SS$)
12: $\text{PNK}$ $\leftarrow$ descendants(trunk) $\cup$ ancestorsUntil($SS$, trunk)
return ($\text{PNK}$, $SS$)
```

Bob sends a **PotentiallyMissing** message encapsulating the output of Algorithm 1. Upon receiving it, and using Algorithm 2, Alice’s device verifies that all epochs referenced in $SS$ exist in $\text{EG}_{Alice}$ (lines 2 - 3). If they do, the local device creates a $\text{toRequest}$ set with all epochs in $\text{PNK}$ that do not exist in $\text{EG}_{Alice}$ and have not already been requested from any device (lines 5 - 7). At this point, Alice discovers all epochs of $\text{EG}_{Bob}$ missing from $\text{EG}_{Alice}$. If the $\text{toRequest}$ set becomes empty, the synchronization session terminates with a **NONEToREQUEST** message, otherwise the session continues to the second phase (Section III-B2).

**Algorithm 2** Determine Missing Epochs

```
1: procedure $\text{MISSINGEpochs}(\text{EG}, \text{toRequest}, \text{SS}, \text{PNK})$
2: for each $e \in \text{SS}$ do
3: if $e \notin \text{EG}$ then return ($\emptyset$, $\text{False}$)
4: $\text{toRequest}$ $\leftarrow \emptyset$
5: for each $e \in \text{PNK}$ do
6: if $e \notin \text{EG}$ $\land$ $e \notin \text{toRequest}$ then
7: $\text{toRequest}$ $\leftarrow$ $\text{toRequest} \cup e$
return ($\text{toRequest}$, $\text{True}$)
```

In case that $\exists e : e \in \text{SS} \land e \notin \text{EG}_{Alice}$ (line 3), Alice cannot proceed with the current synchronization session and terminates it, by sending a **SESSIONABORT** message. This odd situation can happen when Alice chooses to work on a device that lags behind her own other devices in knowledge. However, using information from $SS$ as reported by Bob, she can schedule a new session that starts at a higher point in $\text{EG}$. To do this, she uses Algorithm 3. She iterates $SS$, as received by the **PotentiallyMissing** message (line 3) to populate a known set of epochs $K$. The algorithm inserts in $K$ each known epoch as in (line 5), whereas for any other epoch it first decodes the epoch identifier, to recover the device that created this epoch, and then retrieves the latest known epoch created by this device (line 7). This is possible because one of Alice’s devices has created epoch $e$, $\forall e \in SS$ (Algorithm 1, line 8) and all her devices can decode identifiers created inside her personal group. The latest known epochs of Alice’s devices are accessible because each device in $O^3$ also maintains locally the latest known epoch for each device of the local user and for each share, which are updated when new epochs arrive during a synchronization session. Finally, the algorithm returns the nearest trunk of $K$ (line 7). $\text{EgSync}$ uses this trunk node as the current knowledge $CK$ to initiate a new synchronization session. This second instance of $\text{EgSync}$ protocol’s first phase leads to the discovery of missing epochs, as for each $e \in SS$, it holds that $e \in \text{EG}_{Alice}$ (thus executing lines 5 - 7 of Algorithm 2). With this approach, a device of Alice that lags behind her other devices can catch up with information for this specific share from devices of other users.

**Algorithm 3** Discover new CK

```
1: function $\text{DiscoverCK}(\text{EG}, SS)$:
2: $K$ $\leftarrow$ $\emptyset$
3: for each $e \in SS$ do
4: if $e \in \text{EG}$ then
5: $K$ $\leftarrow K \cup e$
6: else
7: $K$ $\leftarrow K \cup \text{latestKnownEpoch}(e.\text{creator})$
return nearestTrunk($K$)
```

2) Second phase – missing epochs retrieval: In the second phase of the protocol, $\text{toRequest}$ set is requested via a $\text{GETMISSING}$ message and Bob’s device replies with multiple $\text{EPOCHDATA}$ messages, one for each requested epoch. The synchronization session is completed when Alice receives all requested epochs by Bob.

**C. Consistency and privacy properties**

The MS, synchronized via our epoch-graph-based synchronization protocol, forms a delta-state Conflict-Free Replicated data type ($\delta$-CRDT, see [2]) and guarantees Strong Eventual Consistency (SEC, [37]). $\text{EgSync}$ provides SEC because, provided that all messages are delivered, the MS repository will be identical across all nodes, and a conflict can never occur for the MS data itself; the only potential conflicts in our system are for user files which are presented to user-space consistently by our FS layer. For synchronization of objects in our FS (file content), $O^3$ achieves Eventual Filter Consistency (EFC), as defined in [32]. EFC guarantees that if the user defines a replication rule, stating that a set of files $S$ should be replicated to device $D$, then any file $F \in S$ eventually arrives at device $D$, provided that messages are delivered.

We observe that our synchronization protocol can be seen as a causally-ordered broadcast channel (see $\text{cbcast}$ in [7]), where
the messages are epochs. Indeed, if an epoch $A$ happened-before epoch $E$, $A$ will be an ancestor of $E$ in the epoch graph of the specific share. Our MS synchronization protocol ensures that $A$ will always be replayed before $E$, ensuring causal-replay of all dependent epochs. For concurrent epochs, our sync protocol makes no guarantees of ordering. In our MS design, this is immaterial, as the worst that can happen is that different children (versions) of a single node in an object’s version graph might be added, at concurrent epochs. In the replay of these epochs, some devices may place one child before the other, while others may do the opposite. However, the order of children in our epoch graphs is not important, and that’s why our MS repositories are equal across nodes in our system. Due to space, we omit the formal proofs of safety and liveness and refer the reader to the extended version of this paper [21].

To recap, EGSync achieves the following consistency properties:

1) The whole MS (metadata repository and update-log) for a user’s private and shared files is replicated across all of her own devices.
2) For shared files, Alice’s devices can relay epochs created by Bob to Bob’s devices, which have not yet received them.
3) Strong eventual consistency and causality-preserving replay of update actions for the MS repository.

EGSync also achieves the following privacy properties:

1) All non-shared user files are stored locally on a user’s personal devices or, in the case of shared files, on the user’s personal devices as well as those of users with which they collaborate; no third party storage infrastructures are involved.
2) Only shared portions of the MS are replicated across devices of other users.
3) Sharing of existing files does not expose their version history to other users.
4) By hiding device ids as well as counters used to produce identifiers, a user cannot learn the number of devices used by another user, the specific device she used to create a file version, and the amount of work she performs in other private folders that are not shared. We note that epoch graphs do reveal the number of devices a user uses concurrently, but that is not necessarily equal to the number of devices they own and use, in total.

IV. IMPLEMENTATION

We implement the prototype of $O^3$ using C++ on Linux. We use 64-bit identifiers for all objects in the system; we encode in them the device using 4 bits, the user using 12 bits (which is locally translated from the text representation using a dictionary), and use the remaining 48 bits for a number that gets assigned from counters. This 64-bit size is an implementation choice, which can easily scale to 128-bits and more, as needed.

For the MS, we employ a custom data store (DS) using memory-mapped files, copy-on-write mechanisms for checkpointing, and an implementation of B+tree for dictionaries, such as the contents of directories.

The DS maps a sparse data file in memory and provides a memory manager abstraction for allocating and deallocating portions of it. It expects a notification before the calling application modifies any region, with which we implement copy-on-write. $O^3$ uses the DS to store all object, version, and epoch data. We also use a journal to record all high-level DS mutating actions, and DS provides for checkpoints where a new data file and journal file are allocated. Upon checkpoint, $O^3$ is free to continue immediately, while asynchronously the previous data file is flushed to disk and only then the previous journal file is deleted. Upon startup, any existing journal files are replayed to ensure consistency of our data files. We employ this design to avoid fsync calls in the hot-path of the FS’s operations and only perform them in background tasks, allowing us to gain the performance demonstrated in the benchmarks. This is the same approach used by contemporary file systems, such as ext4, where the journal is not fsync’ed whenever written to, but still consistency is guaranteed upon playback. For more details on this mechanism, and on our CS implementations, we refer the interested reader to the extended version of this paper [21].

We implement the FS over FUSE. Implementations compatible with FUSE exist for both Microsoft Windows and Apple Mac OS, giving us confidence about the portability of $O^3$. Furthermore, we recognize that the FS is simply one interface to $O^3$, and different interfaces exist for mobile platforms, such as the Android Storage Access Framework (SAF) and the iOS FileProvider framework. To this end, we have also ported $O^3$ to Android by implementing a SAF provider as its interface, demonstrating that $O^3$ is portable to all major mobile and desktop platforms.

For our asynchronous communications layer, we employ a messaging backend in C++ which abstracts away all node connectivity issues, and manages message submission, delivery, and presence notifications. The network interfacing part of $O^3$ is implemented using event handlers. In the prototype we use for the evaluation in this work, we use direct TCP connections and configuration files to manage node discovery. We describe our work to allow connectivity of users across the Internet in a separate article [4].

V. EVALUATION

We experimentally evaluate $O^3$ to gauge its scaling potential to support offline collaboration. We also conduct file system micro benchmarks to demonstrate that $O^3$’s performance is comparable to the state-of-the-art, when used as a local (versioning) filesystem. We conduct our experiments using a cluster of 7 machines, connected over a Gigabit Ethernet switch. They comprise Octa-core Intel Xeon E5-2620v4 @ 2.10GHz, 32GB RAM, and two 300GB SAS disks, running CentOS 7 Linux. To select workloads that reflect real use cases, the members of our team recorded the file operations inside their
home directories, for an aggregated period of one month, using Linux’s *inotify* service. We use the recorded traces to select the parameters of the Filebench experiment (presented below), and to create a synthetic mixed workload (with both read and write operations) for $O^3$’s scaling experiments. For the network experiments, we simulate a WAN environment by using netem [15], to inject a uniform latency of 50ms for each network packet exchanged. For all experiments we report an average of 3 runs, where standard deviation among runs is reported when higher than 5%.

| Benchmark | $O^3$ | Loopback | Ori |
|-----------|-------|----------|-----|
| create    | 16951.23 | 22657.43 ±7.33% | 19121.08 |
| delete    | 13331.92 | 6462.4 | 4729.69 |
| listdirs  | 6428.44 | 7598.44 ±13.35% | 6748.77 |
| makedirs  | 11109.97 | 3822.13 | 11910.47 |
| removedirs| 12498.77 | 218.29 | 14093.73 |
| stat      | 3570.67 | 26483.39 | 42897.71 ±10.18% |

TABLE I: Filebench Microbenchmarks. Creation, deletion and stat of 200,000 files and creation, deletion and listing of 200,000 directories in a directory tree, reported in ops/sec.

We use Filebench [39] to compare $O^3$ with Loopback (Ext4 over FUSE), as the base of a file system’s performance when operating under FUSE, and Ori [24], a state-of-the-art multi-device, fully replicated file system. We report the results in Table I. Both $O^3$ and Ori outperform Loopback in creation and deletion of directories, since they do not create physical directories but update their data structures, while in listing of files they use cached file metadata. Ori outperforms $O^3$ in the latter, as it caches whole file paths instead of file names, thus achieving constant lookup time. However, this design choice does not allow rename operations. In the deletion of files, $O^3$ outperforms the remaining systems because it does not delete the actual content, as per its versioning semantics, and it simply registers a new file version denoting the operation. In the remaining results (create, listdirs, stat) the systems show comparable performance, taking into account the high standard deviation among runs for Loopback and Ori. Note that, whenever a file is modified and closed, $O^3$ creates a new version, in contrast to the remaining systems. Ori’s versioning functionality is triggered only when the user takes a snapshot of the file system as a whole; thus while Filebench is running it does not track any versions at all.

We now measure the performance of $O^3$ when a user performs file I/O on multiple shares, and plot the results in Figure 5. In the first experiment, we measure the time to execute a mixed workload of 1.2 million file operations (approx. 60% writes/deletes and 40% reads), while distributing them evenly across multiple shares. This workload is the result of recording all file accesses to the home directories of several members of our research team, for an aggregated period of one month, and thus represent a real world usage scenario.

Recall that the operations of each share are recorded in different EGs. We observe that $O^3$’s performance is not affected by the number of EGs it manages in the **MS update log** (blue line in the figure). In the second experiment, we synchronize a second node with all metadata updates. While the total number of updates is the same, more synchronization sessions are required when increasing the number of shares (one synchronization session per share), which $O^3$ handles with sub-linear performance overhead (red line in the figure).

![User scale experiment - WAN](image)

**Fig. 6:** We measure metadata synchronization time in a WAN setting, for a mixed workload distributed evenly across multiple users, when each user initiates a synchronization session to everyone else.

Next, we measure the scaling of **EGSync** when multiple users collaborate concurrently offline. We create one share, we evenly distribute the mixed workload of the previous experiment to $n$ users, we replay the portion of the workload at the corresponding users, and then measure the time it takes for all users to concurrently obtain each other’s work. Each user initiates a synchronization session to every other user in the share, resulting in $n \times (n - 1)$ concurrent synchronization.
sessions in the network. Assuming we have a workload of size $s$ uniformly distributed across a total number of users $n$, each user creates updates of size $s/n$ that are propagated to $n - 1$ users resulting in a total load of $s \times (n - 1)$ on the network. Consequently, when increasing the number of users, more data is transferred over the network. Additionally, we note that this is a worst-case scenario, where all users perform a bulk of work concurrently and then they all attempt to synchronize simultaneously. In Figure 6 we observe that, even in this worst case, $O^3$ scales linearly when synchronizing multiple users concurrently, as expected. Note that, to implement this experiment we used 7 physical nodes each running up to 4 instances of $O^3$. Numbers are accurate since disk I/O does not affect our measurements in this experiment. Also note that we repeated this experiment without WAN simulation but numbers were almost identical, hence we omitted them for brevity. Our synchronization protocol is mostly one-way and thus latency has minimal effect in this experiment; only bandwidth is important.

VI. RELATED WORK

Public and Private Cloud Storage. Popular cloud storage providers ( [3], [10], [13], [25]) offer a number of convenient features such as storage, anytime-anywhere access, automated file synchronization, storage maintenance. Unfortunately, these features come with a significant privacy cost, as users inevitably hand over to the provider their data, or/and their social and working links to other users. Moreover, users may unexpectedly lose their data in case their vendors go out of business [1], or decide to erase old data without warning [18]. Private cloud systems such as ownCloud [27] and network-attached storage products (e.g., [9], [22], [36]) deploy a private server where the user’s data are stored. Such systems avoid the privacy risks of public cloud storage, but are centralized and thus exhibit single points of failure. In addition, they cannot take advantage of opportunistic peer-to-peer connections between devices, instead requiring devices to have continuous connectivity to the private server and to synchronize through the server. They also offer limited support for sharing amongst multiple users, with minimal to no support for conflict detection and resolution. They also limit the number of files that can be stored to the capacity of the private server. Finally, the user shoulders the burden of configuring and maintaining the server.

Versioning Systems. Fossil [31], Venti [30], Elephant [35], Wayback [8] and VersionFS [26] support versioning on a single machine and thus never face conflicts, while $O^3$ supports versioning across different devices and users and provides additional services such as replication and conflict management.

Lazy replication systems. Several systems support lazy, or optimistic, replication including Perspective [34], Cymbiosys [32], Practi [5], Ori [24], Eyo [38], and IMDFs [20].

Perspective and Cymbiosys support cross-device data management for a single user’s data collection. They support partial replication and synchronize data by utilizing user-defined rules. Perspective is a FUSE-based file system, while Cymbiosys is a distributed storage system. Unlike $O^3$, these systems do not provide a global, unified view of all files nor do they fetch files on demand. Moreover, their design is based on flat, non-hierarchical name spaces, making it infeasible to support the bulk of contemporary OSes. In contrast, $O^3$ provides a hierarchical namespace, and thus can be deployed now to support all major mobile and desktop platforms. Moreover, Cymbiosys requires at least one device to hold the user’s entire data collection, while $O^3$ does not.

Eyo and IMDFs support partial replication of a user’s data across devices using user-defined rules. $O^3$ has a “metadata-everywhere” model similar to that of these systems; i.e., $O^3$ replicates its metadata store across all devices. However, Eyo is a storage system (not a file system), and as such, requires all applications to be modified to use it. While Eyo provides a global view of all files, it does so at the cost of treating all objects in isolation (via a flat, non-hierarchical organization). Additionally, both Eyo and IMDFs use synchronization protocols based on version vectors that are not designed to provide support for sharing amongst users in a privacy-preserving manner, thus rendering them single-user solutions. In contrast, $O^3$’s epoch-graph-based synchronization protocol is designed to be used across devices of different users without revealing information about a user’s devices or the progress she makes in her own device group.

Ori is a fully replicated distributed file system that synchronizes data between devices. Ori depends heavily on techniques used in Git [12], a distributed version control system. Ori requires full replication of the data repository across all devices and as a result, cannot integrate space-limited devices (such as smartphones) nor can it support distributed storage whose aggregated capacity is greater than any single device. Moreover, Ori enforces early conflict resolution to restrict users from maintaining inconsistencies on different devices. In contrast, $O^3$ supports both partial replication of data to integrate space-limited devices and provides a global, unified view of both shared and non-shared (private) files, and allows users to resolve conflicts when convenient.

Several systems have provided a subset of the features of $O^3$, in various contexts, such as AFS [17], Ficus [14], Coda [19], Bayou [40], EnsemBlue [28], Anzere [33], Podbase [29]. However, none provides the full functionality of $O^3$. $O^3$ provides a holistic solution that integrates support for a single user’s cross-device personal data management with support for sharing and collaboration with other users, each with their own set of personal devices and needs.

VII. CONCLUSIONS

Many critical Internet services on which users depend are controlled by centralized corporate entities, leading users to trade in their privacy (data or/and affiliations with others) for service. $O^3$ allows users to build private storage clouds for decentralized collaboration with each other, without having to relinquish their data or affiliations to third parties. Our
evaluation shows that $O^3$’s novel EGSync protocol, providing causally-consistent replay of changes across devices and across users, can reasonably scale to support collaboration of users in a completely decentralized manner, without the need for intervention and prying eyes of a third party. Future work includes investigating how $O^3$ could serve as a foundation on which to build other applications and systems, including decentralized social networks, blogging platforms, and multi-media communication applications.

REFERENCES

[1] C. Albanesius, “Megaupload Shutdown Targets Pirates ... And Legitimate Files,” January 2012. [Online]. Available: http://www.pcmag.com/article2/0,2817,2399134,00.asp
[2] P. S. Almeida, A. Shokr, and C. Baquero, “Delta state replicated data types,” Elsevier JPDC, vol. 111, pp. 162 – 173, 2018.
[3] Apple, “Apple Inc. - iCloud,” July 2020. [Online]. Available: https://www.icloud.com/
[4] C. Aslanoglou, M. Konstantopoulos, N. Chondros, and M. Roussopoulos, “Take back your friends with dc:s; A decentralized connectivity service for private social communication apps,” in 2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS). IEEE, 2020, pp. 133–138.
[5] N. Belaramani, M. Dahlin, L. Gao, A. Nayeate, A. Venkataramani, P. Yalagandula, and J. Zheng, “Practi replication,” in USENIX NSDI, 2006.
[6] M. Belotov, “Box.com Leak Discovered,” December 2016. [Online]. Available: https://bestsecuritysearch.com/box-com-leak-discovered/
[7] K. P. Birman, Reliable Distributed Systems, 1st ed. Springer, 2005.
[8] B. Cornell, P. A. Dinda, and F. E. Bustamante, “Wayback: A user-level versioning file system for linux,” in USENIX ATC, 2004.
[9] W. Digital, “My Cloud, Western Digital Personal Cloud Storage,” July 2020. [Online]. Available: https://www.wdc.com/products/personal-cloud-storage/my-cloud.html
[10] Dropbox, “Dropbox Inc. Dropbox - simplify your life,” July 2020. [Online]. Available: https://www.dropbox.com
[11] S. Gibbs, “Dropbox hack leads to leaking of 68m user passwords on the internet,” August 2016. [Online]. Available: https://www.theguardian.com/technology/2016/aug/31/dropbox-hack-passwords-68m-data-breach
[12] Git, “Inc Software Freedom Conservancy, Git,” July 2020. [Online]. Available: https://git-scm.com/
[13] Google Inc., “Google Drive - A safe place for all your files,” July 2020. [Online]. Available: https://drive.google.com
[14] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page, Jr., G. J. Popek, and D. Rothmeier, “Implementation of the Ficus replicated file system,” in USENIX Conference Proceedings. Anaheim, CA: USENIX, Jun 1990.
[15] S. Hemminger et al., “Network emulation with netem,” in Linux conf au, 2005, pp. 18–23.
[16] P. Holand, “Cloudbleed bug: Everything you need to know,” February 2017. [Online]. Available: https://www.cnet.com/how-to/cloudbleed-bug-everything-you-need-to-know/
[17] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, “Scale and performance in a distributed file system,” ACM TOCS, vol. 6, no. 1, Feb 1988.
[18] S. Keane, “MySpace reportedly loses 50 millions songs uploaded over 12 years,” March 2019. [Online]. Available: https://www.cnet.com/news/myspace-reportedly-loses-50-million-songs-uploaded-over-12-years/
[19] J. J. Kistler and M. Satyanarayanan, “Disconnected operation in the coda file system,” ACM TOCS, vol. 10, no. 1, pp. 1–29, Feb 1992.
[20] M. Konstantopoulos, P. Diamantopoulos, N. Chondros, and M. Roussopoulos, “Distributed personal cloud storage without third parties,” IEEE TPDS, vol. 30, no. 11, pp. 2434–2448, Nov 2019.
[21] M. Konstantopoulos, N. Chondros, and M. Roussopoulos, “A social link based private storage cloud,” Oct. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4121108
[22] LaCie, “LaCie CloudBox,” July 2020. [Online]. Available: http://www.lacie.com/gb/en/products/network-storage/cloudbox/
[23] M. Lecuyer, G. Ducotte, F. Lan, A. Papanece, T. Petsios, R. Spuhn, A. Chaintreau, and R. G. Beanbiaw, “Xray: Increasing the web’s transparency with differential correlation,” in USENIX Security Symp, 2014.

ACKNOWLEDGMENT

This work has been supported, in part, by the EU Horizon 2020 NEANIAS Project, Grant #863448 and by ERC Starting Grant #279237.