A Name Is Not A Name: The Implementation Of A Cloud Storage System

Vinh Tao
Scality and Inria-LIP6
vinh.tao@lip6.fr

Vianney Rancurel
Scality
vianney.rancurel@scality.com

João Neto *
KTH and Inria-LIP6
joaon@kth.se

Abstract

The automatic resolution for resolving conflict updates in cloud storage services has been well studied, however, how to correctly implement the resolution in real-world systems remains challenging. In this paper, we present the challenges we experienced when implementing our cloud storage system. They include (1) detecting the intended object for an update when the intended object has been automatically changed by the conflict resolution, and (2) producing no different intermediate results when resolving the conflict updates from more than two replicas. We present our solution of using the mechanism of the conflict resolution to redirect an update to its intended object and of using Conflict-Free Replicated Data Type (CRDT) for a "clean" implementation of conflict resolution without different intermediate results.

1. Introduction

A cloud storage system, such as Dropbox [8], is essentially a distributed file system that uses the eventual consistency approach [23–25] to ensure the availability of its service in the presence of network partition or delay by relaxing the consistency requirement following the CAP theorem [5, 9]. In such a system, all updates are committed locally on each replica before being synchronized with the other replicas. When a conflict happens, i.e., when some updates from different replicas concurrently target the same object, an eventually consistent distributed file system automatically resolves the conflict using its own conflict resolution. For example, when users on different replicas A and B concurrently update the same file `foo`, one of the updates will be preserved in another file, such as "`foo (B's conflicted copy)`" as with Dropbox, to resolve the conflict.

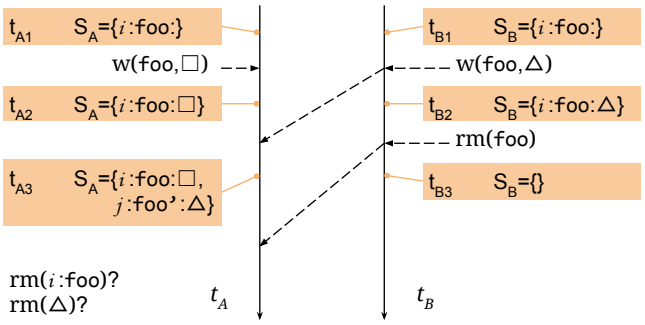

Figure 1. An example challenge for implementing a conflict resolution in eventually consistent distributed file systems. Arrows for timelines; dashed arrows for updates and propagation; S_A and S_B are states at some point in time of replicas A and B, respectively.

The automatic conflict resolution for conflict updates in eventually consistent distributed file systems have been well studied [2, 4, 13, 19, 22], however, how to correctly implement the resolution in real-world systems remains challenging. Consider the previous example (Fig. 1) when users on A and B concurrently write □ and △, respectively, to `foo` of inode i; the user on B later then deletes `foo` after propagating the previous update to A and before receiving the update from A. When receiving B's first update, A would see the conflict on `foo` and would then resolve this conflict by generating another file, named `foo.B` of inode j, to store the update from B. When the second update from B arrives, A would find that it has to delete the file `foo` of inode i. Up to this point, there is an obvious problem: the file which should be deleted in the intention of B's second update is `foo.B` on A, however, all of the information received from B was to delete `foo` of inode i. Implementations of the conflict resolution in this case must therefore identify the intended objects of an update to which an update should be redirected.

Another example is our experiment with Dropbox when we created a system of three replicas A, B, and C and we concurrently updated a common file `foo` on these replicas.

* The work was done when the author was in an internship at Scality.

APSys 2015, July 26–28, 2015, Tokyo, Japan.

http://dx.doi.org/10.1145/2797022.2797034

In some cases on A, Dropbox firstly renamed `foo`, which was containing A's update, to `foo.A`[1] and created another `foo` to store C's update when A received it from C. When A later then received the update of B, Dropbox again renamed `foo`, which was containing C's update, to `foo.C` and created another `foo` to store B's update. From this observation we see that Dropbox, when resolving the conflicting updates, created an intermediate result, i.e., `foo` stored C's and finally B's update in this example. This intermediate result though does not affect the eventual consistency of the replicas, it would however cause some more complexity for conflict resolution if users on A made more updates to `foo` when `foo` was storing C's update.

In this paper, we present the challenges we experienced while implementing conflict resolution for our cloud storage system and we describe our solution for these challenges. These challenges include capturing the intention of those subsequent of a conflict update, and the problem with intermediate conflict resolution results. To the best of our knowledge, we are the first to publicly and systematically formalize and target the problem of implementing conflict resolution in eventually consistent distributed file systems.

We will iterate through possible conflict cases in eventually consistent distributed file systems and corresponding resolution in Section 3. We present our formalization of all problems with automatic conflict resolution in Section 4 and our solution for a correct implementation in Section 5.

2. Existing Approaches

Distributed storage systems that target automatic resolution for conflict updates usually mix between merging semantics in conflict resolution and conflict resolution implementation. The problems of subsequent updates and different intermediate results have not been discussed in these systems neither.

2.1 Distributed File Systems

Locus distributed file system [26] and its descendants such as Ficus [19], Rumor [12], or Roam [18] use another way to resolve the *data conflict* and *naming conflict* that is to move the files in conflict into a special directory and notify users about this automatic decision. This approach, while can converge the replica, interrupts the work on the files on all replicas when a conflict happens. It is not known whether subsequent updates could be directed to the files, which has been moved into another location, or not. Reconciling multiple replicas in these systems is done by pairwise merging without any guarantee about the convergence of the replicas.

Other file systems such as Coda [13], DSF-R [4], and Unison [2] do not specify the way their implementation work with multiple replicas. Ramsey's algebraic approach [17] remains an impractical model as it requires global synchronization, which is a moment when all sites stop and exchange their updates; this model is not the case of real-world systems. Present cloud storage systems such as Microsoft OneDrive [15], Google Drive [11], and Dropbox [8] use propriatery software; these systems can solve the issue with subsequent updates to an automatically renamed file but they also produce different intermediate results.

Versioning file systems, such as Ori [14], explicitly store concurrent updates to the same file in different branches or different checkpoints in the history of the file systems. These systems do not have the problem with subsequent update because the causal relationship between subsequent updates is explict; manual mergings of the concurrent updates to the same file, if required, do not usually generate new files as with systems with automatic conflict resolution. Versioning file systems therefore do not have the problem of subsequent updates of the other optimistic distributed file systems.

The problem of subsequent updates to objects or files in conflict has been casually mentioned in Bayou [23], which is an early distributed system of the eventual consistency approach, and in TierStore [7], a more recent optimistic replication distributed file system. As contrast to Ficus or Coda, users in Bayou and TierStore can still read from and write to objects which are in conflict. The subsequent updates problem, even though has been acknowledged, however was not discussed in detail and was not solved; this is explicitly stated in Bayou: "Of course, the potential drawback of this approach is that newly issued Writes may depend on data that is in conflict and may lead to cascaded conflict resolution." and it remains a question in TierStore: "once the name conflict occurs [...], if the user were to write some new contents to `/foo`, should the new file contents replace both conflicting mappings or just one of them?" We differ from the work in Bayou and in TierStore by directly targeting the problem of subsequent updates in our research and by having a systematic approach for solving the issue.

2.2 Other Systems

Version control systems such as Git [10] or SVN [1] could be also viewed as file system synchronizer when they synchronize the namespaces of projects. Version control systems share the same design of versioning file systems in which concurrent updates are explicitly stored in different branches of a project[2] and merging is done manually. Version control systems therefore do not have the problem of subsequent updates as with eventually consitent distributed file systems with automatic conflict resolution.

Modern NoSQL key-value stores, such as Amazon's Dynamo [6] or Riak [3], have the definition of multiple values for a key which is similar to our system model for conflict resolution if we consider a name in our system as a key. However, these stores are much simpler than file systems in which each version of the value of a key could only be ac-

[1] We use the short names `foo.A`, `foo.B`, and `foo.C` to represent the actual long names that Dropbox created.

[2] Orthogonal updates can be merged automatically but merging these updates generates no new files.

cessible through the key; in file systems, however, these versions must also be exposed to users under their own names, e.g., in new files when resolving *data conflicts*.

3. Conflict Cases and Resolution

In this section, we describe the possible conflict cases and the conflict resolution in eventually consistent distributed file systems. Interested readers can find more about our system model and conflict resolution strategy in another work [22].

We model a file system as a partially ordered set (poset) (Fig. 2). In this model, an element is a directory, a file, or an inode. There are two types of relation between these elements, namely, hierarchy, which is the relationship between a directory and it children, and mapping, which is the relationship between a directory or a file and its inode. The hierarchy relationship is *one-to-many* while the mapping relationship is *one-to-one* for directories and *many-to-one* for files. Because of the *one-to-one* property of the mapping relationship for directories, we consider a pair of a directory and its inode as a single element in the system model.

Concurrent updates that target the same elements in the system model may cause conflict if merging them violates any of these above specifications of the file system model. For example, users on different replicas of a file system concurrently create the same file `foo`; this violates the mapping relationship when a name `foo` is mapped to many inodes. Based on our file system model, we detect the following cases of conflict: *state conflict*, *data conflict*, *naming conflict*, and *mapping conflict* (Fig. 3). We present each of them in the following with their corresponding conflict resolution.

State Conflict This conflict happens when users on different replicas concurrently delete and update an element (a directory or a file). This kind of conflict is resolved by keeping the updated replica in the final result of conflict resolution.

Data Conflict This kind of conflict happens on files only when users concurrently write to the same file inode. There are several ways of resolving this kind of conflict, such as merging the contents into a single file when the file type is known to be mergeable, storing the updates in different versions of the file as with the Copy-On-Write technique, and creating different files to store different updates. In this research and in popular cloud storage systems [8, 11, 15], we choose to use the last solution, which is to create different files to store different contents. Specifically, we remove the original file after creating the new files. For example, when resolving concurrent writes from A and B to the same file `foo`, we make new files `foo.A` and `foo.B` to store the updates from respective replicas, and we remove `foo`.

Naming Conflict This conflict happens when users on different replicas concurrently create different elements (directories and/or files) with the same name. For example, a user on A creates a file `foo` while another user on B creates another directory `foo`. In the case these elements are both file, we rename the files to different names to distinguish them; if they are a file and a directory, we only rename the file; if they are both directory, we recursively merge the contents of these directories.

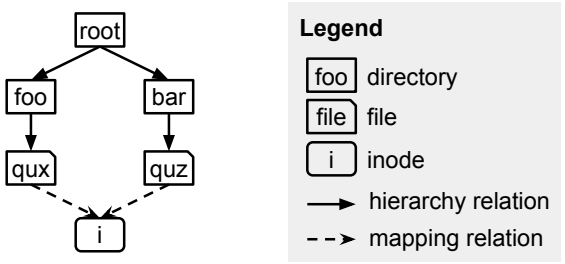

Figure 2. The system model of a file system showing a namespace and its mapping with the file inode. This is a simplified model in which the mapping of a directory and its inode is represented by a single vertex.

Mapping Conflict This conflict happens when users on different replicas concurrently rename the same directory to different names; merging these updates would violate the *one-to-one* property of mapping relation for directories when multiple names are mapped to the same directory inode. We resolve this conflict by making different copies of the directory inode, each of which has the corresponding name. For example, if a directory `foo` is renamed to `bar` and `qux` concurrently, we will make different copies of `foo` and its subtree and rename these copies of `foo` to `bar` and `qux`.

4. Implementation Challenges

The conflict resolution, while resolving conflict updates, makes automatic changes to the file system. This is an implementation issue because the conflict resolution in a real-world system is usually performed asynchronously while the system is still actively working; users can make subsequent updates to an object on a replica while that object's identity, such name or inode number, on another replica has been changed by the automatic conflict resolution. In the cases of *data conflict* and *naming conflict*, subsequent updates targeting the original file cannot find the target on the other replicas when its names has been changed. In the case of *state conflict*, the deleted element is restored by the conflict resolution; this may also cause issue for subsequent updates on the replicas where the object is deleted.

Another implementation challenge is to make the conflict resolution to work with a large number of replicas without introducing any different intermediate state. For example, a system can resolve on A the concurrent updates to file `foo` from B by making new files, such as `foo.A` and `foo.B`, to store the updates of A and B, respectively; the problem then becomes how to identify the concurrency and how to solve it when A receives another concurrent update from C. Some systems such as Dropbox solves this problem by making only one new file, such as `foo.A` in this example, and keep the original name `foo`. However, the problem with

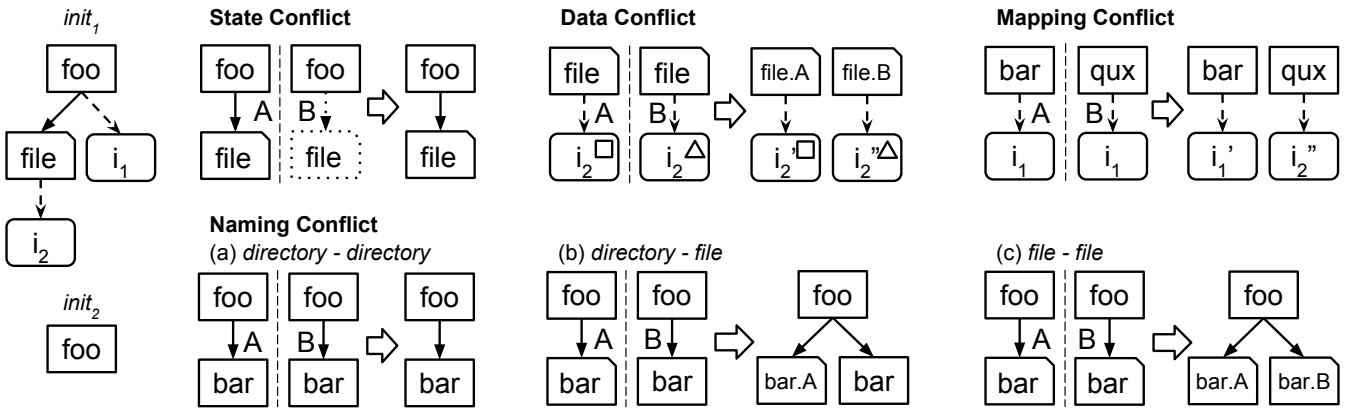

Figure 3. Conflict cases in eventually consistent distributed file systems and our resolution. With $init_1$ and $init_2$ as the starting common state for replicas A and B in the cases of *State Conflict*, *Data Conflict*, and *Mapping Conflict* and in the case of *Naming Conflict*, respectively. Modified states of the replicas are on the left and the final outcomes are on the right of the large arrows.

these approaches is that the original file could go through some different states before coming to the final state. These approaches thus expose intermediate results to users while resolving the conflict of a large number of replicas. In addition, conflict resolution may require coordination between replicas or a centralized service to decide the order of the replicas to keep their updates in the original file.

Based on the conflict cases and resolution and on these above intuitions, we formalize the challenges that we found in implementing the conflict resolution in the following:

1. The intended targets for an update have been changed in automatic conflict resolution. It is important to identify the intended targets for a subsequent update and redirect it to the proper targets.
2. A real-world system usually has a large number of replicas working concurrently. How to identify and resolve the conflict of an arbitrary number of replicas without introducing any different intermediate result is challenging.

5. Our Solution and Implementation

5.1 Solution Overview

Subsequent Updates We use the conflict resolution function to redirect the subsequent updates of a conflict update to their targets. The intuition behind this decision is that conflict resolution generates new files and redirect respective conflict updates to these new files, thus the best solution for redirecting subsequent updates of a conflict update to its intended targets is to use the mechanism of the conflict resolution function. We consider the conflict resolution as a function f that creates new objects and redirects respective updates to these new objects based on its own mechanism. In our example when users on A and B concurrently issue writes operations w_A and w_B, respectively, to `foo`, the conflict resolution function f would create new files `foo.A` and `foo.B` and would redirect the respective updates w_A and w_B to these new files.

In order to be able to redirect subsequent updates, f stores the information about the object on which a conflict happened and a conflict has been resolved. When f receives subsequent updates targeting this object, the function would identify that there are new objects created to store the updates from corresponding replicas and would redirect these subsequent updates to the correct new objects. In our example, f, after resolving conflict on `foo`, marks `foo` as deprecated, which means conflict of w_A and w_B on `foo` has been resolved. When another update rm from B which targets `foo` comes, f would see that `foo` is deprecated and that f should redirect this update to the file `foo.B`. By using the redirecting mechanism of the conflict resolution, the system could find the intended target for a subsequent update.

Multiple Replicas Without Different Intermediate Results To make the conflict resolution to work with an arbitrary number of replicas without generating *different* intermediate result is to make the intermediate results to be subsets of the final result. This is because the "no intermediate result" is only achievable in global synchronization, which requires all replicas to stop and exchange their updates. Real-world system use asynchronous pairwise synchronization between replicas so intermediate results must be a part of the conflict resolution. We can, however, make these intermediate results subsets of the final result to reduce the complexity of exposing them to users. We formalize this as in following:

$$f(a,b) \in f(f(a,b),c) \tag{1}$$

where a, b, and c are concurrent updates to the same object.

We use Conflict Free Replicated Data Type (CRDT) [20, 21]—a specification for eventually consistent data types—to achieve the above result. CRDT ensures the conflict resolution function to be *idempotent*, *commutative*, and *associative*. This means merging any pair of conflict replicas computes their Least-Upper-Bound (LUB), i.e., $f(f(a,b),c) = LUB(f(a,b),c)$ which in turn implies $f(a,b) \in f(f(a,b),c)$.

This is our desired result to make the conflict resolution function to produce no different intermediate result.

In our example of merging on A concurrent updates to `foo` from other replicas B and C, merging the updates from B on A generates $\{$`foo.A`, `foo.B`$\}$; when A receives another update from C, it generates another file `foo.C`. Finally, the state of the replica A is $\{$`foo.A`, `foo.B`, `foo.C`$\}$, which is a superset of the above intermediate state.

5.2 Conflict Resolution Model

Model We model the system as a key-value store in which a vertex, i.e., a directory, a file, or an inode, is an entry whose key is the absolute path (of a directory or a file) or the inode number (of an inode). A key in this model is ensured to be unique sinced an absolute path of a directory or a file and an inode number is assumed to be unique.

For an entry of a directory or a file, the key is the name[3] and the associated value is a set of properties: *inode*, *type*, *state*, and *version*. The property *inode* is the inode number associated with the name, *type* identifies whether the entry is a directory or a file, *version* stores the partial order between different updates from different replicas of the name, and *state* indicates the state of the name, which is either *present*, *deleted*, or *merged*. A name has a state of *present* when it is created and has not been deleted; a name is in the state of *deleted* when it is deleted by a user; a name's state is changed to *merged* when either a *data conflict* or *naming conflict* on it has been resolved. A name in the *merged* state appears to be similar to *deleted* to users in which users cannot see it and can create new directory or file with the name.

The version of an entry increases monotonically. It increases when it is created or when it receives the first update from users since it has been propagated to the other replicas or when it receives an update from another replica. The version is usually implemented in the form of a version vector, which is a popular technique, to store the partial relationship between different updates on different replicas of the entry.

The operations in a file system may change the properties of an entry. For example the operations when a user deletes a file `foo` and then creates another directory `foo` are translated into the state of the entry `foo` is firstly changed to *deleted*, then to *present* and the type of the entry is changed into directory and its inode is changed to a new inode number.

The conflict resolution operation on an entry may change the entry's state into *merged* depending on the type of the conflict. In the case the conflict is either *data conflict* or *naming conflict*, the conflict resolution would change the entry's state into *merged* and would create new entries. For example when users on different replicas A and B concurrently update the same file `foo`, merging these updates on any replicas would change the state of `foo` into *merge* and would create new names `foo.A` and `foo.B`. The properties of these new names are the properties of the replicas of `foo` on A and B, respectively, except the inode numbers. In this case `foo`.$state = merged$, `foo.A`.$inode = i_1$ and `foo.B`.$inode = i_2$.

How Does This Model Solve The Problems In our view, the original named whose state has been changed into *merged* serves as the point to redirect subsequent updates from different replicas to their intended targets. A subsequent update could be viewed as in a conflict with the original name; the conflict resolution is then used to resolve this conflict which would then generate a new file which overwrites the previously created file for the update from the replica of the subsequent update.

In our first example in Figure 1, when resolving the conflict between concurrent updates to `foo` from A and B on A, our conflict resolution changes `foo`'s state to *merged* and creates `foo.A` and `foo.B`. The properties of these entries are as followings: `foo`.$state = merged$, `foo`.$version = LUB(v_A, v_B)$ with v_A and v_B are versions of the `foo` on A and B respectively, and `foo.B`.$version = v_B$. When A receive a subsequent update from B, which is to remove `foo`, with the version v'_B that $v'_B > v_B$, it detects the conflict between its version of `foo` and the received `foo` from B, and A then apply a resolution for the *data conflict* on `foo`, which is to create another `foo.B`, to resolve this conflict. In this case, the conflict resolution updates the *state* and *version* properties of `foo.B` to *deleted* and v'_B, respectively; this is equal to our desired result of redirecting the subsequent update to its intended target which is `foo.B`.

This model also work with multiple replicas without generating *different* intermediate state. Consider the functions to generate new inode numbers and new names are deterministic, the conflict resolution for any pair of conflict update is *commutative*. Indeed, in the case of *data conflict*, merging a pair of conflict updates from different replicas generates new file names with new inodes on all replicas; with the deterministic generating functions, the outcomes of resolving the conflict in these replicas, which are new file names and new inode numbers, are the same. Similarly, the property *associative* of the conflict resolution is also ensured. Consider the original name after resolving a *data conflict* or a *naming conflict* as a redirect point, merging another update from any other replica is to redirect the update to the corresponding generated file. For example the name `foo` resulted from merging concurrent updates from A and B would be the point to redirect other conflict updates from C, D, E to their corresponding files `foo.C`, `foo.D`, and `foo.E`. This result is repeated on any other resolving order of concurrent updates from these replicas and ensure the associative property of the conflict resolution function.

Because of having the properties of *idempotent* (by using an engineering approach which will be described in the next section), *commutative*, and *associative*, the conflict resolution function is actually a kind of CRDT, which ensures the

[3] We use the term *name* in this section to refer to the absolute path of a directory or a file unless stated otherwise.

eventual consistency of replicas without generating different intermediate results.

5.3 Implementing as CRDT

Implementations of the system should follow the specifications of CRDT, it means, to make the conflict resolution in each case become *idempotent*, *commutative*, and *associative* w.r.t implementations' definition of $\leq$ the partial order. In this section, we will discuss in detail the implementation of our conflict resolution that has the above properties.

Partial order We define the partial order $\leq$ based on the timestamps for the states, i.e., if s^t and s^{t+1} are the states of `foo` before and after an update or a merge, then $s^t \leq s^{t+1}$. We use version vector [16], which is a frequently used technique, for assigning timestamps in our system. A version vector for a replica of an object (a file or a directory) is a sparse vector whose elements represent replicas' identifier and the numbers of the updates that have been made on the object from these replicas. Formally, $vv = \{S_i{:}v_i\}_{i=1..n}$ with a pair of $(S_i{:}v_i)$ denotes a replica identifier and number of updates from that replica. For example, the version vector $vv^t_A = <A{:}1, B{:}1>$ of `foo` on A states that A has committed an update on `foo` and A has received from B another update on `foo`; when A commits another update on `foo`, vv_A becomes $vv^{t+1}_A = <A{:}2, B{:}1>$. For any two version vectors vv_A and vv_B, we have $vv_A \leq vv_B$ iff $\forall(S_i{:}v_i) \in vv_A \Rightarrow \exists(S_i{:}v'_i) \in vv_B$ s.t. $v_i \leq_a v'_i$ with $\leq_a$ is an arithmetic comparison of two integers; $vv_A = vv_B$ iff $vv_A \leq vv_B \wedge vv_B \leq vv_A$; vv_A and vv_B are concurrent, denoted by $vv_A || vv_B$, otherwise. In the previous example, $vv^t_A \leq vv^{t+1}_A$ according to the above definition.

Idempotance We make our conflict resolution idempotent by ignoring updates with less recent timestamps. For example, when A merges the update with timestamp v^t_B from B, it increases the timestamp of `foo` from v^t_A to v^{t+1}_A after the merge such that $v^t_A \leq v^{t+1}_A$ and $v^t_B \leq v^{t+1}_A$. When A receives v^t_B again, A would ignore this update because A's v^{t+1}_A is more recent.

State conflict We always preserve the element which has been edited after the merge with concurrent deletes. For example, if `foo` with version v^t is concurrently deleted and updated on A and B to become v^{t+1}_A and v^{t+1}_B, respectively, the result after merging these sites is version v^{t+1}_{AB} with the content of v^{t+1}_B. This implementation makes the merging semantics commutative and associative because the merge function always prefer the edited state over those concurrent deleted; this resembles the bitwise `OR` operation, which is CRDT by definition.

Data conflict The setup for this example include three sites A, B, and C with a common file `foo` of inode i. Users on these sites concurrently write to `foo`.

We make our conflict resolution commutative in the sense that, a pairwise merge of a pair of replicas on either site must produce the same outcome. In this example, merging replicas of `foo` from A and B must produce the same new files `foo.A` of inode i_1 and `foo.B` of inode i_2 on both A and B. We break down the problem of commutativity into making new names and new inode numbers deterministically. We make new names by adding suffixes which are the names of the site of each replica; this function is deterministic since the site names are predefined. We make new inode numbers by hashing the combinations of the original inode number and the site names; with the same deterministic hash function f and the same inputs $\{i, A, B\}$, the outcomes are the same inode numbers $i_1 = f(i, A)$ and $i_2 = f(i, B)$.

We make our conflict resolution associative following our conflict resolution model in Section 5.2. When A receives the replica from B, it creates $\{\texttt{foo.A} \rightarrow i_1, \texttt{foo.B} \rightarrow i_2\}$[4] and changes the state of `foo` into *merged*. When A receives the update from C after the merge with B, it would identify the conflict of C's version with its own version of `foo` and makes `foo.C` $\rightarrow i_3$. When all other sites have done the same process, they will have the same state of $\{\texttt{foo.A} \rightarrow i_1, \texttt{foo.B} \rightarrow i_2, \texttt{foo.C} \rightarrow i_3\}$, regardless of the order of merging these replicas.

Naming conflict We have some cases for this conflict.

Files with the same name In this example, users on A, B, and C concurrently create the same file `foo` with inodes i_1, i_2, and i_3, respectively. The versions for the names on A, B, and C are v^t_A, v^t_B, and v^t_C, respectively.

The conflict resolution for a pairwise merge is commutative because the function to create new names is deterministic as discussed already. For example, merging updates from A and B would create $\{\texttt{foo.A} \rightarrow i_1, \texttt{foo.B} \rightarrow i_2\}$ on both sites. Following our conflict resolution model, the implementation of this conflict resolution is associative. Indeed, the conflict resolution for a pairwise merge on any replica changes the state of the entry of the original name into *merged*; when another update from the other site arrives, the conflict resolution can identify the conflict in the entry of the original name and generate a new name for the received entry. With the previous example, when A receives `foo` $\rightarrow i_3$ from C, it can identify the concurrency between the received `foo` of version v^t_C and its *merged* `foo` of version v^{t+1}_{AB}; merging these names generates `foo.C` $\rightarrow i_3$.

Directories with the same name In this situation, users on A, B, and C create different directories of inodes i_1, i_2, and i_3, respectively, with the same name `foo`. We merge the mapping tables of the directories in a pairwise merges to make another directory which represents these different directories. The problem becomes choosing deterministically an inode number for the merged directory.

We choose the inode deterministically by using our definition of *inode proxy*. When merging with another directory with the same name from another site, the inode number of

[4] We use $\rightarrow$ to present the mapping relation between a name and an inode.

the local directory will be used as the inode number of the merged directory, while the inode number of the remote directory will be an inode proxy, which acts as a proxy that redirects all accesses to it to the main inode. For example, when merging `foo` $\rightarrow i_1$ and `foo` $\rightarrow i_2$ on A, we choose i_1 as the main inode number for the merged directory on A and i_2 becomes an inode proxy of i_1. Accessing i_2 on A would return the merged directory. This is however reversed on B where i_2 is the inode number for the merged directory and i_1 is an inode proxy. This function is indeed commutative and associative because, on all sites by the end, there are i_1, i_2, and i_3 pointing to the same inode.

File and directory with the same name We only change the name of the file. The other information such as inode number and the name of the directory does not change. This function is commutative and associative.

Mapping conflict We make different copies of the directory to preserve the different names. We keep the history of the original inode to support the associativity. In this example, users on A, B, and C start with the same directory `foo` of inode i and they concurrently rename the directory to `bar`, `qux`, and `quz`, respectively. A pairwise merge between A and B would create on both sites the new inodes i_1 and i_2 with `bar` and `qux` are mapped to these new inodes, i.e., $\{\texttt{bar} \rightarrow i_1, \texttt{qux} \rightarrow i_2\}$. The problem of deterministically making new inode numbers has been well described in the case of data conflict; this function is thus commutative.

The state of the inode i is changed into *merged*. When A receives the update from C, it would see the concurrency between its own version of i and the received version from C. Merging these states would generate a new inode i_3 with `quz` $\rightarrow i_3$. After each replica has applied all updates from the other replicas, these replicas will have the same state of $\{\texttt{bar} \rightarrow i_1, \texttt{qux} \rightarrow i_2, \texttt{quz} \rightarrow i_3\}$. The conflict resolution function is thus associative.

6. Evaluation

We compared our prototype named GeoFS to the most popular public cloud storage system Dropbox to see how our conflict resolution model impacts the implementation of our storage system. We knew that it would not be fair when comparing an industrial level system, which has to handle many real-world cases, to a prototype, which has a lot of assumptions. Therefore, we decided to benchmark the ratios of time and the ratios of bandwidth usage it takes these systems to synchronize their replicas in different cases when the number of files in conflict increases.

The experimental setup included four virtual machines running Ubuntu Server 14.04 LTS as replicas, each of which uses one CPU core and 1GB memory of the host. These replicas were in the bridge networking mode; the Dropbox clients was with the *lansync* mode on. We implemented GeoFS in NodeJS and FUSE; the replicas of GeoFS communicates directly with each other over the HTTP protocol.

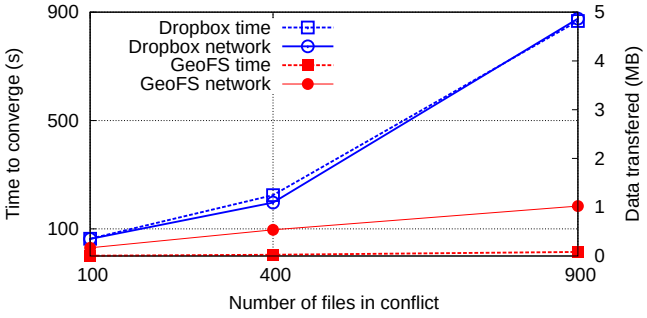

Figure 4. Evaluation of Dropbox and GeoFS. Values are the average for each replica.

On each replica of these systems, we concurrently created n files which are named from 1 to n to create n conflicts for each pair of replicas. The content of each file is the one-byte identifier of the replicas on which the file was created. We tested with different cases when n is 100, 400, and 900.

We measured the time starting from when all files had been created until these systems finished synchronization. For our system, it's easy because we can manipulate the code, however it's difficult for measuring this information in Dropbox system since it's proprietary software could not be interfered. Therefore we decided to use a manual method in which we manually watched the status of all Dropbox's replicas and identified when they finished synchronizing. We used a daemon[5] provided by Dropbox to check the synchronization status, we identified the termination of synchronization only when the status of all replicas is "Up to date"; we used Linux tools `time` and `watch` with an interval of 2 seconds to monitor the replicas' status. In addition, we used another tool `iftop` to track the network usage of both systems.

As shown in Figure 4, GeoFS used much less time and network to synchronize its replicas in all cases compared to Dropbox. More importantly, we see the exponential increase in both the time and the bandwidth that Dropbox used in synchronization as the number of conflicts increased. On the other hand, these measures for GeoFS increased linearly. This can be explained by the synchronization mechanism of each system. As we inspected the network usage of Dropbox, we saw a lot of traffic between each Dropbox replica and `dropbox.com`. Furthermore, Dropbox's synchronation status showed that a replica's status repeatly changed between being stable ("Up to date") then being active ("synchronizing", "uploading", and "downloading") for multiple times before all of these replicas converged. From these observations, we believe that Dropbox uses a form of centralized synchronization where `dropbox.com` acts as the central point. As oppose to Dropbox, our prototype GeoFS is totally decentralized in any sense (network traffic and conflict resolution); this led to the linear time and network usage for synchronization when the number of conflicts increased.

[5] https://www.dropbox.com/download?dl=packages/dropbox.py

7. Conclusions and Future Work

We presented the challenges we encountered when implementing our cloud storage system. They are (1) identifying the intended object of a subsequent update to a conflict object after conflict resolution, and (2) a "clean" implementation method that does not produce different intermediate outcome in merging an arbitrary number of replicas. We target these challenges by applying the framework of CRDT, which is a specification for eventually consistent data types. To the best of our knowledge, we are the first to publicly formalize and target these challenges in a systematic way.

In the future work, we target reducing the complexity that an eventually consistent system exposes to users and developers using a session system. This session system provides a traditional POSIX API while ensuring the eventual consistency of the storage systems. We also target an efficient garbage collection system in future work.

References

[1] Apache. Subversion, v.1.7. https://subversion.apache.org/. Accessed: 2014-12-31.

[2] S. Balasubramaniam and B. C. Pierce. What is a file synchronizer? In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, MobiCom '98, pages 98–108. ACM, 1998.

[3] Basho. Conflict Resolution. http://docs.basho.com/riak/latest/dev/using/conflict-resolution/. Accessed: 2015-04-27.

[4] N. Bjørner. Models and software model checking of a distributed file replication system. *Formal Methods and Hybrid Real-time Systems*, pages 1–23, 2007.

[5] E. A. Brewer. Towards Robust Distributed Systems (Abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[6] 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 Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[7] M. Demmer, B. Du, and E. Brewer. TierStore: A Distributed Filesystem for Challenged Networks in Developing Regions. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 35–48. USENIX, 2008.

[8] Dropbox. Dropbox. https://www.dropbox.com/. Accessed: 2014-12-31.

[9] S. Gilbert and N. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002.

[10] Git. Git. http://git-scm.com/. Accessed: 2014-12-31.

[11] Google. Google Drive. https://www.google.com/drive/. Accessed: 2014-12-31.

[12] R. G. Guy, P. L. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. In *Proceedings of the Workshops on Data Warehousing and Data Mining: Advances in Database Technologies*, ER '98, pages 254–265, 1998.

[13] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. Comput. Syst.*, 10(1):3–25, Feb. 1992.

[14] 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 ACM SIGOPS Symposium on Operating Systems Principles*, pages 151–166, New York, NY, USA, 2013. ACM.

[15] Microsoft. Microsoft OneDrive. https://onedrive.live.com/. Accessed: 2014-12-31.

[16] D. S. Parker Jr, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, (3):240–247, 1983.

[17] N. Ramsey and E. Csirmaz. An algebraic approach to file synchronization. *ACM SIGSOFT Software Engineering Notes*, 26(5):175–185, Sept. 2001.

[18] D. Ratner, P. Reiher, and G. J. Popek. Roam: A scalable replication system for mobile computing. In *Proceedings of the Tenth International Workshop on Database and Expert Systems Applications, 1999.*, pages 96–104. IEEE, 1999.

[19] P. L. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the Ficus file system. In *Usenix Conf.* Usenix, 1994.

[20] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. rr 7506, Inria, rocq, 2011.

[21] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011.

[22] V. Tao, M. Shapiro, and V. Rancurel. Merging Semantics for Conflict Updates in Geo-Distributed File Systems. In *Proceedings of the 8th ACM International Systems and Storage Conference*, Systor '15, New York, NY, USA, 2015. ACM.

[23] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[24] W. Vogels. Eventually Consistent. *ACM Queue*, 6(6):14–19, Oct. 2008.

[25] W. Vogels. Eventually Consistent. *Communications of the ACM*, 52(1):40–44, Jan. 2009.

[26] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. *ACM SIGOPS Operating Systems Review*, 17(5):49–70, 1983.