Merging Semantics for Conflict Updates in Geo-Distributed File Systems

Vinh Tao Thanh, Marc Shapiro, Vianney Rancurel

Merging Semantics for Conflict Updates in Geo-Distributed File Systems

Vinh Tao
Scality, UPMC-LIP6, and INRIA
vinh.tao@lip6.fr

Marc Shapiro
INRIA and UPMC-LIP6
marc.shapiro@acm.org

Vianney Rancurel
Scality
vianney.rancurel@scality.com

Abstract

We present our model of file systems and our merging semantics for resolving conflict updates in geo-distributed file systems. The system model fully describes a file system with all of its components including hard links. This model is able to identify all conflict cases which are classified into direct, such as concurrent updates to the same file, and indirect, such as cycles in the namespace of the file system. The merging semantics resolve all types of conflicts while being able to preserve the effect of all conflict updates. Our implementation of the system and the merging semantics outperforms the existing systems in terms of feature completeness.

Categories and Subject Descriptors D.4.3 [*Operating Systems*]: File Systems Management—Distributed file systems

General Terms Algorithm, Design

Keywords Geo-Distributed File System, Eventual Consistency, CRDT, Conflict Resolution, State-based Replication

1. Introduction

Geo-distributed file systems are those span multiple separate locations, called sites or data centers, each of which fully replicates the state of a common file system. Inter-site networks of these file systems usually have limited bandwidth and high latency compared to the intra-site counterparts. In order to be available and scalable to serve and to adapt to the increasingly high storage demand and large number of users with these networks, geo-distributed file systems usually have to make some trade-off between the consistency and the availability of their services. This trade-off has been formalized in the CAP theorem [Brewer 2000; Gilbert and Lynch 2002].

SYSTOR '15, May 26–28, 2015, Haifa, Israel

http://dx.doi.org/10.1145/2757667.2757683

The eventual consistency approach [Vogels 2009, 2008; Terry et al. 1995] is a usual solution for large scale geo-distributed file systems to solve their problems of availability and scalability. In eventually consistent systems, all updates are committed locally on each site before being asynchronously propagated to the other sites. The eventual consistency approach ensures that, when all sites have received and applied all updates from each other, all sites will have the same state. Because updates are committed locally without coordination between sites, writing to these distributed file systems is fast, adding more sites increases the throughput linearly, and a site could be disconnected while users on a site can still make modifications to the file systems.

The most long-standing issue with eventually consistent systems is that they may have conflict updates when synchronizing the sites. These updates are those from different sites that concurrently target the same part of the file system. For example, users on different sites may concurrently write to the same file; it is difficult to choose which way to merge the updates so that all sites have the same state after merging.

In this work, we focus on the issue of resolving conflict updates in eventually consistent geo-distributed file systems.

1.1 Existing Approaches

The existing approaches to the problem of conflict resolution for synchronization in geo-distributed file system are classified into two groups: *operation-based* and *state-based*.

The operation-based approaches log the file system operations on each site and then propagate the log to the other sites on which these operations may be replayed to keep the replicas consistent. Examples include IceCube [Kermarrec et al. 2001], Bayou [Terry et al. 1995], and Ramsey's algebraic approach [Ramsey et al. 2001].

The state-based approaches keep track of the state of each file and directory, then the final states or deltas of the changed files and directories are propagated to the other sites to be merged there. Examples of this approach are Ficus [Reiher et al. 1994], Coda [Kistler and Satyanarayanan 1992; Satyanarayanan et al. 1990], Unison [Balasubramaniam and Pierce 1998], Andrew File System [Howard 1988; Kazar 1988], and Microsoft's DFS-R [Bjørner 2007].

The operation-based approaches however usually require global synchronizations, which are moments when all sites stop receiving more updates and exchange their logs to define new sequences of operations to be applied on each site; this is not practical in real-world geo-distributed file systems. Moreover, an experience with our deployment [Segura et al. 2014] for a large telecommunications service provider in France has shown that the number of operations in that real-world system is three orders of magnitude larger than the number of changed files and directories, which results in much larger log size as compared to final states or deltas.

The state-based approaches usually do not model file systems completely. For example, these approaches assume that different names in the namespace are mapped to different files, thus they do not work with file hard links in file systems. This incorrect file system model may lead to anomalous behaviours as described in Section 6 of this paper.

1.2 Our Contributions

In this work, we target the problems of resolving conflict updates when synchronizing diverged replicas of a file system. Our contributions are:

- We fully model file systems which include: the hierarchical namespace and the mapping between the namespace and content (Section 2). Our model works with all components, including hard links, and with all operations, such as *rename*, of file systems, which are the limitations of the existing approaches.
- We propose an update detection strategy that helps systematically identifying both direct and indirect conflicts (Section 4.1). This strategy improves the existing ad-hoc approaches in detecting indirect conflicts such as namespace cycles.
- We specify our merging semantics for resolving conflict updates (Section 4.2). These merging semantics, from the user's perspective, preserve both the namespaces of the replicas and the data of these conflict updates.

2. System Model

In this section, we describe our model of real-world file systems and conflict cases in geo-distributed file systems.

2.1 File System

A file system is composed of (1) a namespace that presents a hierarchical structure of the file system to users and (2) the mapping between the names in the namespace and data.

The namespace is a directed rooted tree whose edges point away from the root[1] (Fig. 1a). In this structure, there is a vertex *root* which is the starting point for navigating the tree. For any other vertex v, there is only one directed path p from *root* to v; this path is also referred to as the absolute path of v. Absolute paths are unique, i.e., there are no two vertices with the same absolute path.

Each path[2] specifies the strict total order information over the set of vertices in the path. A path p of a vertex v_i is formalized by:

$$p = \{v_0 \rightarrow \ldots \rightarrow v_i, i = 0..n\} \quad (1)$$

where v_0 is *root* and $\rightarrow$ is the hierarchy relation of the vertices, which is a *one-to-many* relationship.

Real-world file systems strictly follow the above specification, however, they may use different notations. In Microsoft NTFS, the root is the device name, e.g., `C:`, and the hierarchy relation is denoted with a backward slash `\`. In the implementations of the POSIX API, the forward slash `/` is used for both the root and the hierarchy relation. With these notations, the highlighted path in Figure 1a is denoted as `C:\foo\file` in Microsoft NTFS while it is `/foo/file` in POSIX implementations.

For simplicity, we will use the POSIX-style notation, i.e., to use forward slash `/` for both the root and the hierarchy relation, in the remaining parts of the paper to present paths.

The mapping between the namespace and data (Fig. 1b), denoted by $\rightarrow$, is a separate mapping relation between vertices, which are represented by their unique paths $\{p_i, i = 1..n\}$, and data objects (named *inodes*[3]) $\{i_j, j = 1..m\}$.

There are two predefined types of inode: *directory* and *file*. The mapping of vertices to directory inodes is always *one-to-one* while the mapping of vertices to file inodes is *many-to-one*. That means, a vertex maps to only one inode of any type, and many vertices can map to a file inode. These mappings are defined as:

$$\begin{cases} p_i \xrightarrow{1:1} i_j & type(i_j) = directory \\ p_i \xrightarrow{n:1} i_j & type(i_j) = file \end{cases} \quad (2)$$

whereas the function *type* tells whether an inode is of type *directory* or *file*. The type of a vertex is identified as the type of the inode that the vertex is mapped to. A leaf vertex is only referred to as a directory or a file depending on the type of the inode to which the path of the vertex is mapped, while the type of a non-leaf vertex (including the root) is always a directory.

The full file system could be then described by a composition of the hierarchy and mappings relations, as depicted in Figure 1b; this model is known in the literature as a partially ordered set (poset). Because of the *one-to-one* relationship between vertices and directory inodes, we can consider these mappings as a single vertex in the full model of the file system without impacting correctness; the simplified model (Fig. 1c) is still a poset of vertices and inodes.

[1] We argue that the back and self pointers of directories in real-world implementations are an engineering optimization to support the use of relative paths; not modelling these pointers does not impact the correctness of our system model.

[2] Unless stated otherwise, we use the term *path* to refer to the absolute path of a vertex.

[3] We use the term *inode* to refer to an inode object, following POSIX terms, and its referring data as a whole.

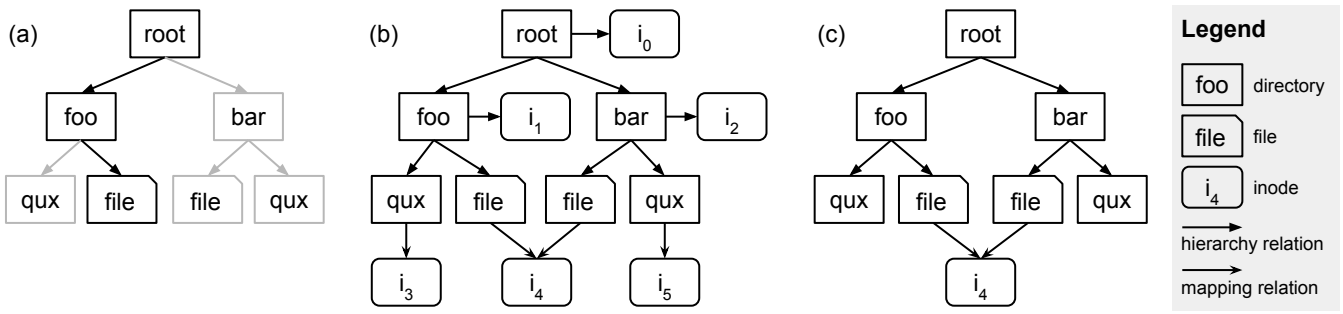

Figure 1. System model showing a namespace and its mapping with inodes. With diagrams a shows the namespace as a rooted-directed tree, b describes the full system model with the namespace and its mapping with inodes, and c depicts the simplified system model where pairs of vertex and directory inode are grouped together into single elements.

2.2 Divergence and Reconciliation

Diverged replicas of a file system may differ in the tree structures of their namespaces, in the mappings of the namespaces and the inodes, or in the content of file inodes. The divergence is a possible outcome of concurrent operations, which are used by eventual consistency in the presence of network partition or delay.

The examples in Figure 2 present some possible cases of divergence. In the first example (Fig. 2a), users on different sites A and B see different namespaces of the file system. In the next example (Fig. 2b), users on both sites A and B see the same namespace with the paths `/foo` and `/bar`. However, due to different mappings, these paths represent different files on A, while on B, they are hard links to the same file. These different mappings lead to the divergence in the structures of the replicas. In another example (Fig. 2c), replicas on both A and B have the same namespace and mappings, but the contents of the file at `/foo` are different due to different uses on these sites.

Reconciling diverged replicas is to bring the diverged replicas of a file system to a common state where they have the same namespace, same mappings, and same file contents. The common state should be 'meaningful' in the sense that it can be explained intuitively to users.

Reconciliation is a pairwise process in which each pair of diverged replicas asynchronously exchanges and synchronizes their states. A pair of diverged replicas is reconciled (or merged) by simply computing the unions of their updated elements, such as vertices and inodes. Merging elements that are updated on either site updates the other; merging those updated on both sites, a situation which is referred to as a *conflict*, may require conflict resolution.

2.3 Conflict Cases

Conflicts are either *commutative* and *non-commutative*. A commutative conflict is a pair of diverged replicas whose merged state is the superset of their states and whose merged state does not violate any specification of the system model. For example, if a user on a site creates a file `/foo` and another user on another site creates a file `/bar`, merging the replicas in this case results in {`/foo`, `/bar`} which is the superset of both replicas and does not violate any specification of the system model. The other conflicts are non-commutative. For example, if a directory `/foo` is concurrently renamed to `/bar` on a site and to `/qux` on another site, then merging the replicas computes their superset, which is {`/bar`, `/qux`}, however, the result violates the *one-to-one* property of the directory vertex-inode mapping. In this research, we only focus on the non-commutative conflicts and use the term *conflict* to refer to those of this type unless stated otherwise.

We classify the conflict cases into two groups: *direct* and *indirect*.

Direct conflicts are caused by concurrent updates that target the same elements, such as vertices, inodes, or mappings. The direct cases are listed bellow.

State conflict (Fig. 3a). An element is concurrently deleted and modified by users on different sites. For example, a file is removed while it is being edited. The different states, *removed* and *updated*, of the same element cannot appear together in the merged result.

Data conflict (Fig. 3b). Users from different sites concurrently write to the same file. These different contents generally cannot be merged together in the same inode (we will consider this issue in more detail in Section 4.3.1).

Naming conflict (Fig. 3c, 3d, 3e). This type of conflict happens when vertices with the same path exist. For example, users on different sites may concurrently create/rename files and/or directories with/to the same name. Merging these replicas violates the uniqueness of vertices, or in another way, violates the *one-to-one* and *many-to-one* properties of the mapping relations when it causes a path to be mapped to many inodes.

Mapping conflict (Fig. 3f). Users on different sites map different names to a directory inode. As a result, merging these replicas violates the *one-to-one* directory vertex-inode mapping. For example, the name of a directory `/foo`, which is mapped to inode i, is concurrently changed to `/bar` and

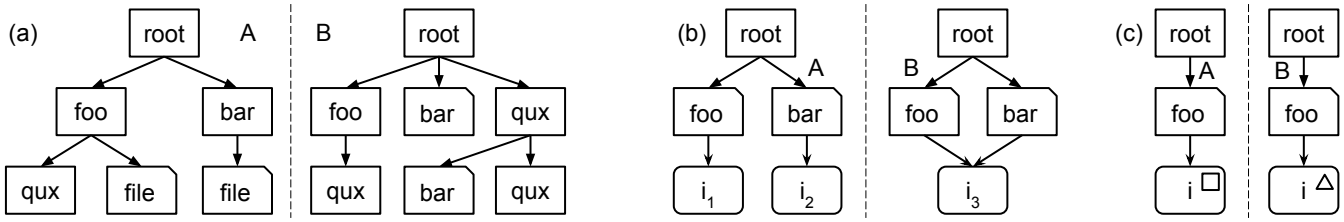

Figure 2. Example of diverged replicas of sites A and B. With examples a, b, and c describe different scenarios of divergence of namespaces, namespace-inode mappings, and inode contents, respectively; square and triangle are different updated contents.

`/qux` on sites A and B, respectively. Merging these updates makes both `/bar` and `/qux` to be mapped to i, which violates the directory mapping rule. Concurrent changes to the name of a file, however, is not an issue because the mapping of vertices to file inodes is a *many-to-one* relationship.

Indirect conflicts are caused by updates that target different elements, such that merging the replicas to which these updates target would not cause any direct conflict, but the merging result would be anomalous. In the followings, we study some common examples used in the literature and formally define indirect conflicts.

Example 1. Delete-while-edit (Fig. 3g). The example illustrates the situation where the file `/foo/file` is edited on site A when the directory `/foo` is deleted on site B. Merging the replicas on A and B would not result in any conflict since the updates target different elements of the system, but in the result, the file `/foo/file` is deleted.

Example 2. Cycles in the namespace (Fig. 3h). In this example, the directory `foo` is moved into another directory `bar` on site A, while on site B, `bar` is moved into `foo`. Merging these replicas produces a directed cycle between `foo` and `bar`, which violates the partial order relationship.

The common pattern of indirect conflicts from these above examples is that updates that target different elements in a file system interfere in the path of each other. This is illustrated in Example 1 where the delete update modifies the path of the updated file, and in Example 2 where each update modifies the path of the other.

3. Existing Approaches

State-based approaches to the problem of conflict resolution for synchronization in geo-distributed file systems can be classified as *namespace approaches* and *inode approaches*.

3.1 The Namespace Approach

The namespace approach models a file system as a collection of paths, each of which represents a different directory or file. Merging the replicas of a file system computes the union of the path collections. Conflicts happen when the same path representing different contents exists. Examples of this approach are Unison [Balasubramaniam and Pierce 1998], Dropbox [Dropbox], and version control systems such as Git [Git] and SVN [Apache b].

The namespace approach is free from the indirect conflicts. In the case of *delete-while-edit*, because the existence of an updated element (file or directory) on a site implies that of its parent directories on the same site, merging by computing the union of the path collections ensures the element and its parent directories to exist in the merge result. In the case of *cycles-in-the-namespace*, because the number of the paths in a collection is finite, merging by computing the union of different collections results in a finite number of paths; therefore cycles cannot form when they need to be represented by an infinite number of paths.

The namespace approach however does not fully model file systems. It assumes the *one-to-one* mapping between paths and inodes, and thus this approach does not take into account hard links. This results in the waste of storage and bandwidth when synchronizing replicas. For example, a new name linked to an existing file could be considered to be a new file, and thus the content of the file could need to be synchronized again between replicas. Another limitation of not modelling hard links is in detecting whether a path has been updated or not. In the case there is an update to a file inode with multiple names, the namespace approach only identifies the names through which the file inode was updated as updated. This approach therefore cannot detect the direct conflict when users on different sites write to the same file inode through different names. Moreover, implementations of an incorrect file system model may result in anomalous outcomes when synchronizing diverged replicas as described in the experiments with Dropbox (Section 6).

3.2 The Inode Approach

The inode approach models a file system as a collection of separate inodes (or database records as in the case of DFS-R [Bjørner 2007]). The namespace is stored in the directory inodes and data is stored in the file inodes. Merging diverged replicas involves computing the union of the corresponding inode collections. This is actually how real-world file systems are implemented. Examples of this approach are LOCUS [Walker et al. 1983; Parker Jr et al. 1983; Popek and Walker 1985] and its descendants, such as Ficus [Reiher et al. 1994] Rumor [Guy et al. 1999] and Roam [Ratner et al. 1999, 2004].

The inode approach however is prone to the indirect conflicts, when updates target elements that are on the same path

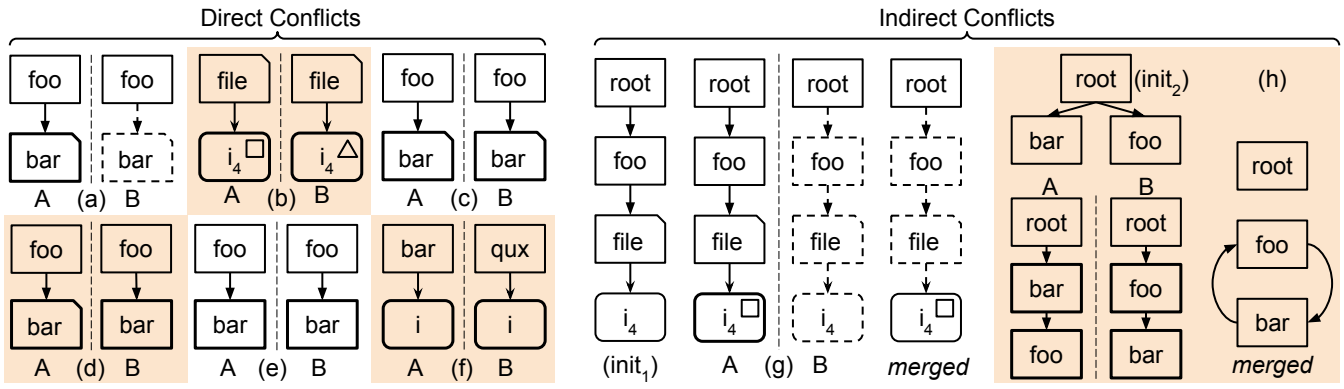

Figure 3. Conflict cases with a, b, c, d, e, and f as examples of direct conflict and g and h as examples of indirect conflict. With A and B are different sites; dashed elements are those deleted; squares and triangles represent different updated contents; bold shapes are updated elements; the diagrams $init_1$ and $init_2$ are the initial common states of the replicas in the examples g and h, respectively; *merged* is the final common state of the file system after merging the diverged replicas.

but are stored in different inodes. Detecting and resolving indirect conflicts are usually ad-hoc. For example in Ficus, the updated files in *delete-while-edit* indirect conflicts are moved into a special directory, or in DFS-R, cycles in the namespace are stored in some arbitrary directories.

4. Conflict Resolution

In this section, we propose our update detection strategy that can systematically identify all conflict cases, and we describe our general principles and the detail merging semantics for resolving these conflicts.

4.1 Update Detection

What is an update? An update is a change in the state of the file system caused by some operations. We classify these operations into two main groups, *deletion* and *modification*. Those of *deletion* type are the operations that remove a vertex (directory, file, or inode) or a relationship (hierarchy or mapping). Those of *modification* are operations that modify existing elements of the system, such as writing to an existing inode or changing the name of a vertex. The creation of a new file or directory is regarded as a modification of the parent directory. The *rename* operation on a vertex is considered a deletion of the vertex's old relationship and a creation of the vertex's new relationship which, as a result, modify the target parent directory and the vertex itself.

What are affected by an update? The direct targets, i.e., vertices or relationships, are the first to be changed by an update; these are the cause for direct conflicts. With our definition of indirect conflicts, i.e., updates that do not directly conflict but target the path of the others, we believe that if we want to detect indirect conflicts, we need to also identify the elements in the path of an updated element as updated. Indeed, by doing so, we would be able to break an indirect conflict into several direct conflicts when an update targets an element in the path of another.

The path of a deleted element, however, is not regarded as being updated. This strategy is to prevent the situation where multiple deletions would not delete a path.

4.2 Conflict Resolution Principles

We believe that the common state of the file system after merging diverged replicas is 'meaningful' to users when the users can still see the effect of their own updates after the merge. We formalize this 'meaningfulness' by the following principles for resolving conflict updates.

Principle 1: No lost update. To preserve all updates on all replicas because these updates are equally valid.

Principle 2: No side-effect. To not cause anomalous merge results. For example, moving `/foo` into `/bar` on a site and `/bar` into `/foo` on another site may cause them to disappear after the merge when a cycle of these directories is formed.

From the above principles, we identify their main properties, they are *element preservation* and *relationship preservation*. These properties are described in the followings.

4.2.1 Element Preservation

The state of an updated replica of an element is always preserved when its conflict with another updated replica of that element is resolved in a pairwise merge.

We define the act of resolving conflicts when merging a pair of diverged replicas of an element as a function, named $merge$, that takes the replicas as inputs and produces an output that preserves the states of these replicas. This function with *element preservation* is formally defined as:

$$merge(r^A, r^B) = r' : r^A \subseteq r', r^B \subseteq r' \qquad (3)$$

where r^A and r^B are diverged replicas of an element on sites A and B, respectively.

We also require that the outcomes of different mergings of replicas of different elements be also different. For example, merging diverged replicas of the file `foo` and merging

those of the file `bar` do not result in the same file `qux`. This requirement is defined as:

$$r_x \neq r_y \iff merge(r_x, r_x^*) \neq merge(r_y, r_y^*) \quad (4)$$

where r_x and r_y are replicas on the same site of different elements x and y of the file system; r_x^* and r_y^* are replicas of x and y on any other site.

The *element preservation* property makes the *merge* function to be able to preserve the consistency of the user's view of their data before and after the merge. This means, if the user updates any file or directory, the updated elements should also be available in the merging outcome. Some examples of using *merge* with *element preservation* in merging conflict updates are described in Figure 4. The first example (Fig. 4a) depicts the situation where the content of a file is changed on different sites A and B; these updates are then preserved by *merge* in different files after the merge. The second example (Fig. 4b) presents the case when a directory `foo` is concurrently renamed to `bar` and `qux` on A and B, respectively; the new names are preserved by mapping them with different copies of the directory inode.

The *merge* function with *element preservation* has the same effect to the Write-Wins strategy, which prefers modifications to other concurrent deletions, in merging concurrent updates to the same element. It preserves an updated replica in the output when merging this replica with another deleted replica of the same element. The remaining problem is to choose a presentation for this solution that does not surprise users with strange merging results, such as a locally deleted file reappearing after the merge.

The *merge* function with *element preservation* fully preserves all of the states of concurrently updated replicas after merging them, as opposed to the Last-Writer-Wins (LWW) approach when LWW only preserves one of these states based on their timestamps. As a result, users of systems using the LWW approach may see their updates to a file, which have been locally committed, become unavailable after merging.

4.2.2 Relationship Preservation

The mapping and the hierarchy relations between updated vertices of a replica of a file system are preserved when merging a replica with another replica. The *merge* function with the *relationship preservation* property is formalized by:

$$\begin{cases} r_v \rightarrow r_i \Rightarrow merge(r_v, r_v^*) \rightarrow merge(r_i, r_i^*) \\ r_x / r_y \Rightarrow merge(r_x, r_x^*) / merge(r_y, r_y^*) \end{cases} \quad (5)$$

where r_v and r_i are replicas on the same site of a vertex v and an inode i, r_x and r_y are replicas on the same site of different vertices x and y of the file system.

This *relationship preservation* property enables *merge* to present a familiar file system structure, which includes the namespace and the vertex-inode mappings, to users on different sites before and after the merge. This means, if users change the structure of the file system, after the merge, they should be able to see the updated structure in the merged result. For example, if a directory `foo` is moved into another directory `bar` on a site, the updated hierarchy relationship `foo/bar` should be preserved when merging with concurrent modifications to these directories on another site.

This property enables *merge* to work with file system hard links in the sense that *merge* can preserve the mapping between a file inode and its names when merging. For example, if users on different sites A and B update an inode *i* through its hard links `foo` and `bar`, respectively, then *merge* would preserve both structures of $\{\texttt{foo} \rightarrow i, \texttt{bar} \rightarrow i\}$ on each site in the outcome. This is an advantage of our system compared to the namespace approach when the latter cannot detect the conflict on `foo` and `bar`.

The *merge* function with both *element preservation* and *relationship preservation* properties is able to resolves all conflicts w.r.t to our principles of 'meaningfulness'. Formally, *merge* embeds the structure of each diverged replica into the final common state, which means *merge* preserves all updates and presents no side-effect. Indeed, each of the file system replicas could be represented as a poset, and merging them would (1) preserve the partial order between elements (Equation 5) and (2) map input and output elements in a *one-to-one* relationship (Equation 3, 4); these specifications are well-known in the field of order theory with the names of *order-preserving* and *order-reflecting*, which are the requirements to embed a poset into another.

An example of using *merge* with *relationship preservation* is shown in (Fig. 4c). In this example, users on A and B concurrently create new hard links `bar` and `qux`, respectively, to file `foo` of inode i_0 and concurrently write $\square$ and $\triangle$ to it. Before the merge, users on A would see $\{\texttt{foo} \rightarrow i_0(\square), \texttt{bar} \rightarrow i_0(\square)\}$, while users on B see $\{\texttt{foo} \rightarrow i_0(\triangle), \texttt{qux} \rightarrow i_0(\triangle)\}$. Then *merge* would preserve these structures when merging in $\{\texttt{foo.A} \rightarrow i_1(\square), \texttt{bar} \rightarrow i_1(\square)\}$ and $\{\texttt{foo.B} \rightarrow i_2(\triangle), \texttt{qux} \rightarrow i_2(\triangle)\}$, respectively.

4.3 Merging Semantics

The merging semantics implement our conflict resolution principles in detail policies. In this section, we describe these policies for all conflict cases.

4.3.1 Merging Direct Conflicts

State conflict A state conflict happens when an element is updated and deleted concurrently on different sites. The merging policy for this kind of conflict depends on whether the element is a directory or a file.

The element is a file: In this case, the updated replica is preserved with the name of the original file. This policy preserves the view of the users on the updated site where they see the same namespace and content after the merge. Users on the site where the file is deleted, however, would see the deleted file to appear again with new content.

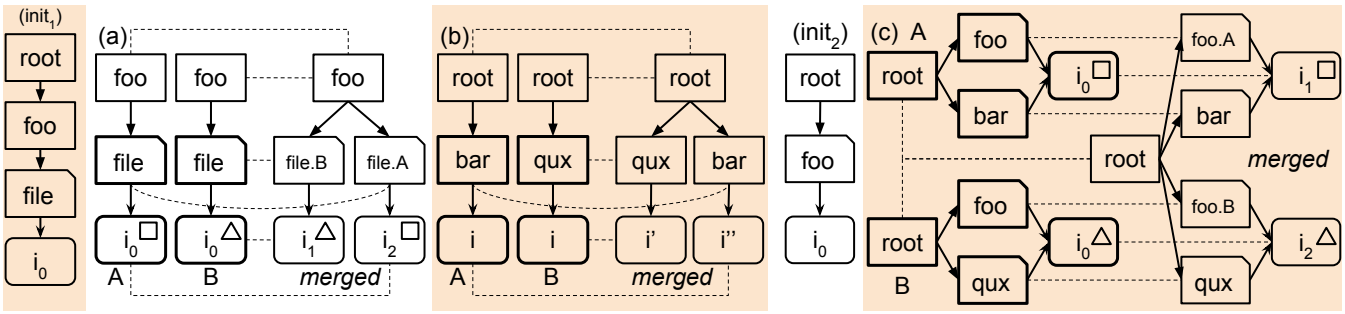

Figure 4. Examples of merging diverged replicas of a file system using the *merge* function with the *element preservation* property and *merge* with the *relationship preservation* property. The diagrams $init_1$ and $init_2$ show the common inital states of the file systems in examples a and b and in example c respectively; A and B denotes the names of different sites; $merged$ is the final common state of the file system after merging the diverged replicas; thin dashed-lines are for the correspondences between the input and output elements of the merge function; bold shapes are updated elements with squares and triangles as different updated contents.

The element is a directory: In this case, the directory is preserved with the same names in the merging result. The idea behind this decision is the same to that of the previous policy. However, the content of the merged element depends on merging its sub-directories and sub-files.

Data conflict A data conflict happens when the content of a file has been updated on multiple sites. We have different policies to resolve this kind of conflict, they are (1) make new files (2) merge the contents and (3) branch the states.

The first policy (Fig. 5a) is to make new files to store the updated contents. The names of these files are chosen so that users on each site would be able to know that there is a conflict and to identify which file has their updates. A traditional approach is to append to the original names a specific identifier such as the unique identifier of the site of each replica. Industrial cloud storage services choose to use more information-rich solutions for generating the new file name, for example, to append "conflicted copy" to the original file name as with Dropbox.

The next policy (Fig. 5b) is to merge the contents together in the same file. This policy is used by the *merge* function when the file is of a data-type that is known to be mergeable. An example of such data-type could be text-based as some recent studies [Preguiça et al. 2009; Weiss et al. 2009] have shown that, merging concurrent updates to the content of text files is feasible.

In the last policy (Fig. 5c), diverged states are stored in different content branches of the file. Users or administrators can merge the branches manually. This pattern is extensively used in version control systems, such as Git [Git] or SVN [Apache b], to manage collaborative software developments. Another usage of this solution is in the content of virtual machine. A virtual machine is represented as a file; concurrent uses of a virtual machine would lead to diverged replicas of the file, which are then merged and represented by different branches using the Copy-On-Write technique. By doing so, multiple users can use multiple virtual machines of the same file without making multiple copies of the file.

Naming conflict This is the situation where different directories and/or files with the same name exist.

Files with the same name We change the names to distinguish different files (Fig. 5a). New names are generated by adding suffixes to the orginal name as in *data conflict* policy.

File and directory with the same name We only change the name of the file (Fig. 5d). This decision is to preserve the namespaces as much as possible, i.e., the paths of the sub-elements of the directory would not have to change.

Directories with the same name They are merged to become a single directory (Fig. 5e); sub-element collections are merged together in which this could be recursively broken down into other *naming conflict* cases.

Mapping conflict This case happens when users on different sites rename a same directory to different names, which violates the *one-to-one* property of directory vertex-inode mapping relationship. We resolve this conflict by recursively make copies of the directory and its sub-elements to preserve the namespaces, which include different names of the directory (Fig. 5f).

We decide, however, not to make copies of the sub-files but to link new names to these files (Fig. 5g). The decision is not only to save storage but also to deal with the situation where a file has multiple names; copying the file would cause these names to be replicated and mapped to different copies, which makes merging more complex.

4.3.2 Merging Indirect Conflicts

Each of the indirect conflicts is a combination of some direct conflicts. In this section, we go through the previously mentioned examples and show how to deal with these problems by using the policies for direct cases.

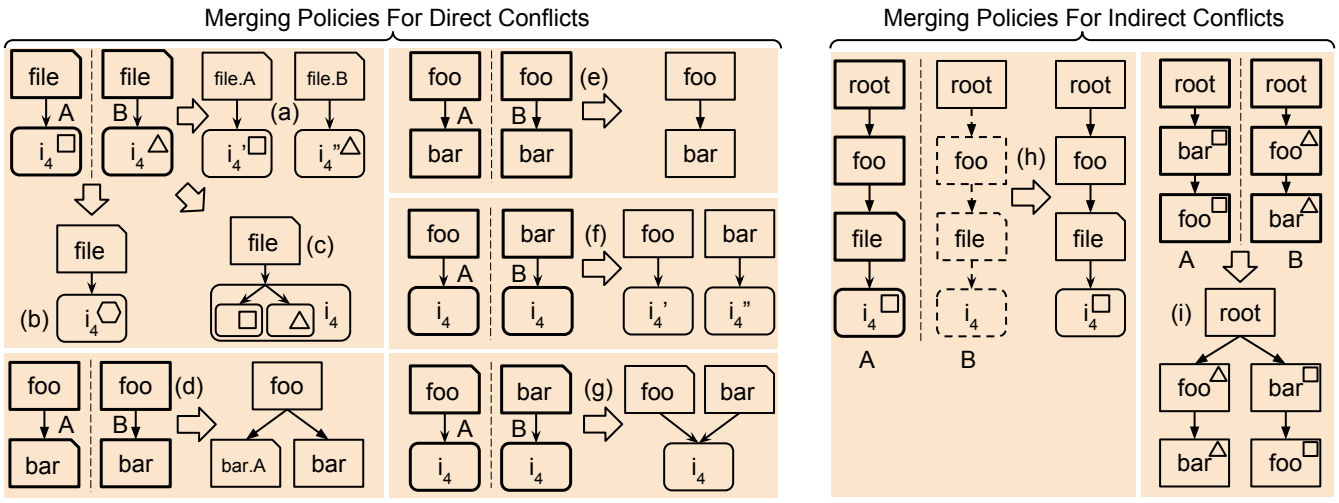

Figure 5. Merging policies for conflicts. We only display the final outcome on each site.

Deleting while editing In this example (Fig. 5h), a file is modified on a site while a directory on its path is deleted on another site. Because each of the elements in the path of an updated element is also regarded as being updated, the problem becomes merging pairs of replicas that are in *state conflict* of elements in the path. By simply applying the policy for *state conflict* for these elements, the path of the updated file and the file itself are preserved after the merge.

Making cycles in the namespace This example (Fig. 5i) can be viewed as a set of other direct *mapping conflict* problems. Because the directory, to which another directory moved, is considered updated, the directories on its path are also viewed as updated. When merging with other renamed replicas, there would be *mapping conflicts* on these directories (`foo` and `bar` in this case). We can recursively solve these conflicts on `foo` and `bar` using the *mapping conflict* policy. Different relationships with different parents of a directory are preserved in its corresponding copies, which are different, after the merge so that cycles are not formed.

5. Implementation Framework

In this section, we describe the specifications of a framework for implementing our merging semantics to support working with an arbitrary number of replicas. This framework is specified based on CRDT [Shapiro et al. 2011a,b], which is a specification for eventual consistency data types.

5.1 CRDT

CRDT, which stands for Conflict Free Replicated Data Type, is a set of specifications for data types to support eventual consistency. We focus on state-based CRDT in this work and summarize the requirements of state-based CRDT as follow.

The state of each replica advances upward after modifications w.r.t a partial order $\leq$. Formally, if s_i and s_j are the states of a replica before and after an update, then $s_i \leq s_j$.

The *merge* function computes the Least Upper Bound (LUB) of these replicas w.r.t $\leq$. The LUB of a pair of states s_i and s_j under the partial order $\leq$ is defined as

$$s = LUB(s_i, s_j) : \begin{cases} s_i \leq s, s_j \leq s \\ \nexists s' \leq s : s_i \leq s', s_j \leq s' \end{cases} . \quad (6)$$

By definition, the LUB function is *idempotent*, *commutative*, and *associative*:

$$\begin{cases} idempotent & LUB(s, s) = s \\ commutative & LUB(s_i, s_j) = LUB(s_j, s_i) \\ associative & LUB(s_i, LUB(s_j, s_k)) = \\ & \quad LUB(LUB(s_i, s_j), s_k) \end{cases} . \quad (7)$$

The *idempotent* property implies that merging the same replica multiple times does not change the result. This property can deal with the unreliable networks where updates could be delivered multiple times. The *commutative* property also works well with these networks; it enables the function *merge* to handle updates in any order. The last property *associative* ensures merging any number of replicas in any order would still make them consistent. This is an important property for CRDT to work with an arbitrary number of sites.

5.2 Implementation Example

The implementation of a system should follow the specifications of CRDT, which means, to make the merging policies become *idempotent*, *commutative*, and *associative* w.r.t the implementation's definition of $\leq$ the partial order. Discussing the detail of our implementation is out of the scope of this paper.

Nevertheless, we describe an example of making the merging policy for *data conflict* to be CRDT in this section. 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 define the partial order $\leq$ based on the timestamps of 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, which is a frequently used technique, for assigning timestamps in our system.

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

We make our merging policy 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 that 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 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 merging policy associative using the versioning system. When A receives the replica from B, it creates $\{\texttt{foo.A} \rightarrow i_1, \texttt{foo.B} \rightarrow i_2\}$ and marks $\texttt{foo} \rightarrow i$ as deleted in merge. When the replica from C comes, A sees the conflict of the replica with its version of `foo` and makes $\{\texttt{foo.A} \rightarrow i_1, \texttt{foo.C} \rightarrow i_3\}$. 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.

6. Compare Features

We compare the features of our approach and the existing approaches represented by commercial systems, which are Dropbox [Dropbox], Google Drive [Google], and Microsoft OneDrive [Microsoft] (Table 1).

We implemented a prototype, named GeoFS, with NodeJS and FUSE. We used processes, communicating over HTTP, as sites. The host machine ran Ubuntu Desktop 14.04 LTS.

The setup of Dropbox was two virtual machines running Ubuntu Server 14.04 LTS with Dropbox client for Linux $v.3.0.3$ as the replicas A and B. They are hosted in the same machine with Network Address Translation networking.

The setup for Google Drive and Microsoft OneDrive was a Mac running Mac OS X $v.10.10$ as site A and a virtual machine of Windows 8.1 Enterprise as site B. These sites were in the same local network. The Google Drive clients for Mac and Windows were the same, $v.1.18.7821.2489$, while the Microsoft OneDrive clients on these sites were $v.17.3.4501$ and $v.6.3.9600.17334$, respectively.

Table 1. Evaluation of our merging semantics with commercial systems. Abbreviations: Db for Dropbox, GD for Google Drive, and OD for Microsoft OneDrive.

Feature/Support	Db	GD	OD	GeoFS
Preserve Updates	✓	×	✓	✓
Preserve Structure	×	×	×	✓
Hard link	×	×	×	✓
Same name dir./files	✓	dvg.[a]	✓	✓
Write || Write	✓	lww[b]	✓	✓
Direct Delete || Edit	✓	d.w.[c]	✓	✓
Indirect Delete || Edit	✓	d.w.	✓	✓
Cycles	✓	arb.[d]	×	✓

[a] Diverge: elements are preserved, but replicas' structures diverged.
[b] Last-Writer-Wins: the write with the last timestamp wins over the others.
[c] Delete-Wins: the element, if deleted on any site, is deleted after merging.
[d] Arbitrary: the directories in the cycles are placed at *root* after merge.

In the following experiments, we determined how well these systems could resolve the conflict cases that we described in Section 2.3. In all of the cases, our prototype was able resolve the conflicts and to produce the desired outcomes of our merging semantics. For such result, we only describe the behaviours of the commercial systems with the experiments in the followings.

Experiment 1. Support for hard links We created a file `foo` and its hard link `bar` on A. After merging, B, in any of the setups, had the same namespace of `foo` and `bar`. However, these files on B pointed to different inodes in all cases, which means there is a divergence in the structures of the replicas.

There were also some anomalous result we observed with Dropbox in this experiment. We updated `bar` on A, after merging the update to B, we deleted both `foo` and `bar` on B. However, `foo` appeared again on all sites after the merge.

Experiment 2. Files and/or directories with same name We concurrently created a file `foo` on both A and B. On each site, these systems created different names for these files to distinguish them. However, the system using Google Drive did not converge the replica to the same structure, i.e., `foo`'s content on A was not equal to `foo`'s on B.

The results were repeated when we created a file on a site and another directory with the same name on the other site. Systems using Dropbox and OneDrive were able to make their replicas converge while Google Drive was not.

Experiment 3. Concurrent writes to the same file We wrote to the same existing file on different site. The systems using Dropbox and OneDrive resolved the conflict by creating different files to distinguish these updated versions of the file. However, the system using Google Drive only retained the update from one site. Based on this result, we believe that Google Drive uses the Last-Writer-Wins approach to resolve

the concurrent writes to the same file which, while simple, does not preserve all the updates.

Experiment 4. Delete while editing We did different experiments with the direct and the indirect cases. In the direct case, we deleted a file `foo` on A while writing to it on B. Except the system using Google Drive which deleted `foo` on both sites after the merge, the other systems preserved the updated file. In the indirect case, we deleted the directory `bar`, whose sub-files were `qux` and `quz`, on A, while on B, we wrote to `qux`. After merging, the systems using Dropbox and OneDrive preserved `qux`, `bar` and deleted `quz`, while the system using Google Drive deleted `bar` and its sub-files.

We assumed that Google Drive resolved this conflict using the Delete-Wins approach, i.e., an element that has been deleted on any replica would be deleted after the merge.

Experiment 5. Cycles in the namespace Initially, A and B started with the same namespace of directories `foo` and `bar`. On A, we moved `foo` into `bar`, while on B, we moved `bar` into `foo` to make the cycle of these directories. We expected the merged namespace would be `foo/bar` and `bar/foo`, which embeds both that of A and B. The system using Dropbox was able to make the expected outcome, while the Google Drive system put all the directories in the cycle in *root* and the system using OneDrive stayed diverged.

As a conclussion, while Dropbox and OneDrive can support preserving updated elements, they do not preserve the structure of the file system when merging diverged replicas.

7. Related Work

Database systems The problem of resolving conflict updates has also been studied in the field of database system. Databases usually model their system as a collection of tables with rows as in traditional relational databases or simply as a key-value store as with modern NoSQL databases. Operations on these systems are either *insert*, *update*, or *delete* at the row or key level. Conflict cases in these systems are therefore mostly situations where a row or a key is concurrently created or updated; these cases are very limited in number as compared to those in file systems. Because of the space constrain, we only present the conflict resolution in Oracle [Oracle], which is a representative example of relational databases, and Dynamo [DeCandia et al. 2007] and Riak [Basho], which are examples of NoSQL databases.

There are update conflict, uniqueness conflict, and delete conflict in Oracle. An update conflict happens when a row is concurrent updated by users on different sites. Oracle resolves this conflict by using either the LWW approach or some additional mechanisms that produce a deterministic outcome for the concurrent updates, such as *additive* that aggregates the update values. A uniqueness conflict, which happens when different rows with the same primary key are concurrently created on different sites, is resolved by adding some sequence, such as site identifier or a number, to the value of the primary keys to make them unique. A delete conflict, which is the case when a row is concurrently deleted and updated on different sites, requires manual intervention.

All of these above conflict cases for relational databases are considered the same conflict in NoSQL databases in which a key is mapped to different values. Dynamo and Riak solve this conflict by using either the LWW approach or by keeping these values as different versions of the key.

Merging framework The problem of merging diverged replicas of a general model has been discussed elsewhere [Pottinger and Bernstein 2003]. Pottinger and Bernstein proposed in this work some merging semantics including *element preservation* and *relationship preservation*, which inspired the formalization of our model. They also presented some resolutions for conflict updates in merging models. For example, conlicts in the *type-of* relationship, which is known to be *one-to-one*, are solved in the same way to our system, in which new types are created. Tree cycles are either collapsed into a single element or need manual intervention. Apart from the domain, we are different from them in the objectives of merging, i.e., we try to preserve the structures of the replicas by using *order-preserving* and *order-reflecting*.

Version control systems Git [Git] and SVN [Apache b] are the representative examples of distributed and centralized version control systems, which could also be viewed as simplified file systems. The main focus of these systems is on keeping replicas of files of different projects consistent by keeping the namespaces synchronized. There could be different versions of the files or different versions of the whole projects at the same time in the system with the name 'branches'. These systems rely totally on manual intervention from users to solve the conflict updates. In version control systems, the operation *rename* is regarded as a combination of *delete* and *create*. These systems do not have the support for hard links.

8. Conclusions and Future Work

In this paper, we presented our theory and implementation of merging diverged replicas of a geo-distributed file system. We proposed our merging semantics which were implemented in detail by merging policies that work with all components and all operations on the file systems. These policies can handle both the direct and the indirect conflicts, in which the latter is known to be the limitation of existing approaches. Finally, we described our specifications for an implementation framework that ensures the eventual consistency and some detail implementation techniques to achieve the desired merging properties. To the best of our knowledge, the model of indirect conflicts and the merging semantics for them are novel in the distributed file system field.

We also target some complement features in future work. They are a session system that reduces the complexity of the eventual consistency approach for users and developers and a study of the feasibility on applying these semantics in modern distributed file systems such as HDFS [Apache a].

References

Apache. HDFS User Guide, v.2.6.0. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html, a. Accessed: 2015-04-27.

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

Sundar Balasubramaniam and Benjamin 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.

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

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

Eric 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.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner 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.

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

Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

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

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

Richard Guy, Peter Reiher, D. Rather, Michial Gunter, Wilkie Ma, and Gerald Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. *Lecture notes in computer science*, pages 254–265, 1999.

John H. Howard. On overview of the andrew file system. In *Proceedings of the USENIX Winter Conference. Dallas, Texas, USA, January 1988*, pages 23–26, 1988.

Michael L. Kazar. Synchronization and caching issues in the andrew file system. In *Proceedings of the USENIX Winter Conference. Dallas, Texas, USA, January 1988*, pages 27–36, 1988.

Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The icecube approach to the reconciliation of divergent replicas. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 210–218. ACM, 2001.

James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst.*, 10(1):3–25, February 1992.

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

Oracle. Oracle Database Advanced Replication, 12c Release 1 (12.1). http://docs.oracle.com/database/121/REPLN/E53117-02.pdf. Accessed: 2015-04-27.

Douglas Stott Parker Jr, Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, (3):240–247, 1983.

Gerald Popek and Bruce J. Walker. *The LOCUS distributed system architecture*. MIT press, 1985.

Rachel A. Pottinger and Philip A. Bernstein. Merging models based on given correspondences. In *Proceedings of the 29th international conference on Very large data bases*, VLDB '03, pages 862–873. VLDB Endowment, 2003.

Nuno Preguiça, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403. IEEE Computer Society, 2009.

Norman Ramsey, El Csirmaz, et al. An algebraic approach to file synchronization. *ACM SIGSOFT Software Engineering Notes*, 26(5):175–185, 2001.

David Ratner, Peter Reiher, and Gerald J Popek. Roam: A scalable replication system for mobile computing. In *Database and Expert Systems Applications, 1999. Proceedings. Tenth International Workshop on*, pages 96–104. IEEE, 1999.

David Ratner, Peter Reiher, and Gerald J Popek. Roam: a scalable replication system for mobility. *Mobile Networks and Applications*, 9(5):537–544, 2004.

Peter L Reiher, John S Heidemann, David Ratner, Gregory Skinner, and Gerald J Popek. *Resolving file conflicts in the Ficus file system*. UCLA Computer Science Department, 1994.

Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

Marc Segura, Vianney Rancurel, Vinh Tao, and Marc Shapiro. Scality's experience with a geo-distributed file system. In *Proceedings of the Posters & Demos Session*, Middleware Posters and Demos '14, pages 31–32, New York, NY, USA, 2014. ACM.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011a.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski, et al. A comprehensive study of convergent and commutative replicated data types. 2011b.

D. B. Terry, M. M. Theimer, Karin 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.

Werner Vogels. Eventually consistent. *Queue*, 6(6):14–19, October 2008.

Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.

Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The locus distributed operating system. *ACM SIGOPS Operating Systems Review*, 17(5):49–70, 1983.

Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 404–412. IEEE, 2009.