# Summer Scholars Technical Report

Constantin Berzan Norman Ramsey

## 1 Introduction

Current methods for synchronizing file systems are either limited to two replicas, or depend on a central server to maintain an authoritative version of the data. We aimed to develop a multi-replica synchronizer that does not exhibit either of these limitations.

The synchronizer is based on the algebra of file system operations defined by Ramsey and Csirmaz [1]. A working file system is modeled as a finite map from paths to files or directories. The algebra defines the following file system operations:

| | |
|---|---|
| $create(\pi, X)\ F$ | creates file or directory $X$ at $\pi$ in file system $F$ |
| $edit(\pi, \textsc{Dir}(m))\ F$ | edits $F$ at $\pi$ to be a directory with metadata $m$ |
| $edit(\pi, \textsc{File}(m, x))\ F$ | edits $F$ at $\pi$ to be a file with contents $x$ and metadata $m$ |
| $remove(\pi)\ F$ | removes the file or directory at $\pi$ in $F$ |
| $break\ F$ | breaks the file system $F$ |

The commands $edit(\pi, \textsc{Dir}(m))$ and $edit(\pi, \textsc{File}(m, x))$ are defined separately because they have different preconditions. A broken file system is denoted $\perp$, and is included in the model to enable reasoning about invalid sequences of operations. Once broken by an invalid operation (such as creating a file that already exists), the file system cannot be unbroken by any sequence of operations.

## 2 Results

### 2.1 Policy Variations

Reconciliation policies vary according to which operations they propagate in the presence of conflicts. The *liberal* policy is at one extreme: it propagates an operation to all replicas where it does not break the file system. The *conservative* policy is at the other extreme: it never propagates an operation under a conflicted directory. There exist *intermediate* policies: for example, we could devise a policy that propagates commands under conflicted paths, but only if they can be propagated to all replicas.

#### 2.1.1 An Intermediate Policy

We will first show that the original reconciler by Ramsey and Csirmaz [1] implements an intermediate policy. The reconciler computes $S_j^*$, the sequences of commands to propagate to a given replica, like this:

$$S_j^* = [\, C \mid i \in 1..n, C \in S_i, C \,\text{should be propagated to replica}\, j]$$

A command $C \in S_i$ should be propagated to replica $j$ iff:

- $C \notin S_j$ ($C$ has not already been performed at replica $j$);
- no commands at replicas other than $i$ conflict with $C$;
- no commands at replicas other than $i$ conflict with commands that must precede $C$.

Conflicting commands are defined in terms of minimal sequences from the update detector. Commands $C_i \in S_i$ and $C_j \in S_j$ (where $i \neq j$) *conflict* iff $C_j \notin S_i$ and $C_i \notin S_j$ and one of the following is true:

- $C_i; C_j \not\Vdash C_j; C_i$, i.e., the commands do not have a common upper bound.
- $C_i; C_j \equiv break$ or $C_j; C_i \equiv break$, i.e., the commands break every file system.

A command $C'$ *must precede* command $C$ iff they appear in the same sequence $S_i$, $C'$ precedes $C$ in $S_i$, and they do not have an upper bound ($C'; C \not\Vdash C; C'$).

Looking closely at this definition, we found that $C'$ *must precede* $C$ was incompatible with $C'$ and $C$ belonging to the same minimal update sequence, so we revised the definition of "must precede" thus: $C'$ *must precede* $C$ iff they appear in the same sequence $S_i$, $C'$ precedes $C$ in $S_i$, and their reversal is not an improvement: $C'; C \not\sqsubseteq C; C'$.

Finally, we can examine the policy that this reconciler implements. Consider this example:

| replica A | | replica B | |
|---|---|---|---|
| $C_1$ | $edit(/a, \text{DIR}(m_1))$ | $C_3$ | $edit(/a, \text{DIR}(m_3))$ |
| $C_2$ | $create(/a/b, \text{FILE}(m_2, c_2))$ | | |

Here, $C_1$ *must precede* $C_2$, which prevents $C_2$ from being propagated to replica B. Therefore an operation under the conflicted path */a is not* propagated. Another example:

| replica A | | replica B | | replica C | |
|---|---|---|---|---|---|
| $C_1$ | $edit(/a, \text{DIR}(m_1))$ | $C_2$ | $edit(/a, \text{DIR}(m_2))$ | $C_3$ | $create(/a/b, \text{FILE}(m_3, c_3))$ |

Here $C_3$ conflicts with neither $C_1$ nor $C_2$, nor is there any command in sequence $S_C$ that must precede $C_3$. Therefore $C_3$ *is* propagated to replicas A and B, under the conflict at */a*.

Since this reconciler propagates some operations under a conflict, but not others, it implements an intermediate policy.

### 2.1.2 Another Intermediate Policy

One would expect that removing the "must precede" check from the previously presented reconciler makes it liberal. However, this is not the case. Without the "must precede" check, we are left with the following algorithm for computing the $S_j^*$ and $S_j^{\oslash}$ sequences:

$$\begin{aligned} R_j^* &= \{\, C \mid i \in 1..n, C \in S_i, C \notin S_j, \\ &\qquad \nexists C' \in S_k \text{ s.t. } C \notin S_k \wedge C' \notin S_i \wedge C' \oslash C \} \\ S_j^* &= [\, C \mid C \in R_j^* ] \text{ in canonical order} \\ S_j^{\oslash} &= [\, C \mid C \in S_j, \\ &\qquad \exists C' \in S_k \text{ s.t. } C \notin S_k \wedge C' \notin S_j \wedge C' \oslash C ] \end{aligned}$$

Consider this example:

| replica A | | replica B | |
|---|---|---|---|
| $C_1$ | $edit(/a, \text{DIR}(m_1))$ | $C_3$ | $edit(/a, \text{DIR}(m_3))$ |
| $C_2$ | $create(/a/b, \text{FILE}(m_2, c_2))$ | | |

Here $C_2$ can be propagated to replica B ($C_3; C_2 \not\equiv break$), and it *does* get propagated (no conflict between $C_2$ and $C_3$). Another example:

| replica A | | replica B | |
|---|---|---|---|
| $C_1$ | $create(/a, \text{DIR}(m_1))$ | $C_3$ | $create(/a, \text{DIR}(m_3))$ |
| $C_2$ | $create(/a/b, \text{FILE}(m_2, c_2))$ | | |

Here $C_2$ can be propagated to replica B as well ($C_3; C_2 \not\equiv break$), but it *does not* get propagated ($C_2 \oslash C_3$ since $C_2; C_3 \equiv break$).

Thus this reconciler also implements an intermediate policy.

### 2.1.3 The Conservative Policy

We modified the reconciler of Ramsey and Csirmaz [1] to follow the conservative policy. Paths at which commands conflict are added to $P^{\oslash}$, and a command at path $\pi$ is propagated only if neither $\pi$ nor any of its ancestors is in $P^{\oslash}$.

$$P^{\oslash} = \{\, path(C) \mid i \in 1..n, C \in S_i, \exists C' \in S_j \text{ s.t. } C \notin S_j \wedge C' \notin S_i \wedge C' \oslash C \}$$
$$R^*_j = \{\, C \mid i \in 1..n, C \in S_i, C \notin S_j \wedge (path(C) = \pi/\pi' \Rightarrow \pi \notin P^{\oslash}) \}$$
$$S^*_j = [\, C \mid C \in R^*_j] \text{ in canonical order}$$
$$S^{\oslash}_j = [\, C \mid C \in S_j, path(C) = \pi/\pi' \wedge \pi \in P^{\oslash}] \ (\pi' \text{ may be empty})$$

Using QuickCheck, we have verified that this version of the reconciler satisfies a number of postconditions. Let the file system at last synchronization be $F$, and the current file system at each of the $n$ replicas be $F_i$. Let $S_i$ be the sequence of updates detected at each replica, such that $S_i F = F_i$. Let $S^*_i$ and $S^{\oslash}_i$ be the propagated and conflicting sequences from the algorithm above. Then the following hold:

1. $S^*_i F_i \not\equiv \bot$ (applying the propagated sequence at a replica does not break it).
2. If all $S^{\oslash}_i$ are empty, then all resulting file systems $S^*_i F_i$ are identical (the synchronization makes all replicas identical).

Let $F'_i = S^*_i F_i$ for each $i$ (the propagated sequences are applied at each replica). Let $S'_i$ be the updates detected from the original $F$, i.e. $S'_i F = F'_i$, and suppose the reconciler runs again, with outputs $S'^*_i$ and $S'^{\oslash}_i$. Then the following hold:

1. All $S'^*_i$ are empty (the reconciler finds nothing more to propagate).
2. $S'^{\oslash}_i = S^{\oslash}_i$ for all $i$ (the reconciler finds the same conflicts as last time).

### 2.1.4 The Liberal Policy

Upon attempting to implement the liberal policy, we discovered two things.

First, the postconditions of the conservative reconciler do not hold. Some of the extra changes that get propagated become conflicts in the second run of the reconciler, so the sets of detected conflicts change.

Second, it is impossible to implement the liberal policy as a function with the same signature as that of the conservative policy:

$$reconcile :: [UpdateSeq] \to [(PropagateSeq, ConflictSeq)]$$

Consider this example:

| replica A | | replica B | | replica C |
|---|---|---|---|---|
| $C_1$ | $edit(/a, \text{DIR}(m_1))$ | $C_3$ | $edit(/a, \text{DIR}(m_3))$ | (no changes) |
| $C_2$ | $create(/a/b, \text{FILE}(m_2, c_2))$ | | | |

Here $C_2$ should be propagated to replica C only if $/a$ is a directory on that replica, but the function has no way to obtain that information.

## 2.2 Theoretical and Algorithmic Developments

### 2.2.1 A Tree-based Conservative Reconciler

We developed an alternative, tree-based algorithm implementing the conservative policy:

1. Make a tree out of all paths mentioned in all $S_i$, and all their ancestors. ($T$ is a tree with nodes representing paths in the filesystem.)

2. Label the tree with the operations from $S_i$. (Each node in $T$ is labeled with a finite map Ops from replica IDs to operations, and two additional boolean flags: AssertDir and PropagateConflictsUp. The flags are initially False.)

   For a command $C \in S_i$ at $\pi$, the tree is modified as follows:

   (a) add $i \mapsto C$ to Ops at $\pi$.
   (b) if $C = create(\pi, X)$, then set AssertDir at $parent(\pi)$ to True.
   (c) if $C = remove(\pi)$ or $C = edit(\pi, \text{File}(m, c))$, then set PropagateConflictsUp at $\pi$ to True.

3. Traverse the tree in postorder and mark each node conflicted or not. A node at $\pi$ is conflicted iff one of the following:

   (a) Ops at $\pi$ contains two distinct commands;
   (b) $\pi$ is labeled AssertDir, Ops is not empty, and a command in Ops is $remove(\pi)$, $edit(\pi, \text{File}(m, c))$, or $create(\pi, \text{File}(m, c))$;
   (c) $\pi$ is labeled PropagateConflictsUp, and one of its children is marked conflicted.

4. Traverse the tree again and compute the sequences $S_i^*$ and $S_i^{\oslash}$.

$$S_i^* = [\, C \mid C \notin S_i, C \text{ the unique operation in } \text{Ops}(\pi), \text{ neither } \pi \text{ nor any of its ancestors conflicted}]$$
$$S_i^{\oslash} = [\, C \mid (i \mapsto C) \in \text{Ops}(\pi), \pi \text{ or one of its ancestors conflicted}]$$

5. Put sequences $S_i^*$ and $S_i^{\oslash}$ in canonical order.

We proved that this algorithm produces the same results as the conservative sequence-based algorithm described earlier.

### 2.2.2 Binary Relations

We defined a set of binary relations such that for any possible pair of operations in the algebra, exactly one relation applies:

| | notation | | effect of $C_1; C_2$ | | effect of $C_2; C_1$ | name |
|---|---|---|---|---|---|---|
| 1. | $C_1 \overset{b}{\sim} C_2$ | iff | $C_1; C_2 \equiv break$ | and | $C_2; C_1 \equiv break$ | (bothBreak) |
| 2. | $C_1 \overset{b}{\succcurlyeq} C_2$ | iff | $C_1; C_2 \equiv break$ | and | $C_2; C_1 \not\equiv break$ | (swapUnbreaks) |
| 3. | $C_1 \overset{b}{\preccurlyeq} C_2$ | iff | $C_1; C_2 \not\equiv break$ | and | $C_2; C_1 \equiv break$ | (swapBreaks) |
| 4. | $C_1 \lessdot C_2$ | iff | $C_1; C_2 \not\equiv break$ | and | $C_2; C_1 \not\equiv break \wedge$ $C_2; C_1 \sqsubseteq C_1; C_2 \wedge$ $C_2; C_1 \not\equiv C_1; C_2$ | (swapWorsens) |
| 5. | $C_1 \gtrdot C_2$ | iff | $C_1; C_2 \not\equiv break$ | and | $C_2; C_1 \not\equiv break \wedge$ $C_2; C_1 \sqsupseteq C_1; C_2 \wedge$ $C_2; C_1 \not\equiv C_1; C_2$ | (swapImproves) |
| 6. | $C_1 \sim C_2$ | iff | $C_1; C_2 \not\equiv break$ | and | $C_2; C_1 \not\equiv break \wedge$ $C_2; C_1 \equiv C_1; C_2$ | (commuteNoBreak) |
| 7. | $C_1 \bowtie C_2$ | iff | $C_1; C_2 \not\equiv break$ | and | $C_2; C_1 \not\equiv break \wedge$ $C_2; C_1 \not\parallel C_1; C_2$ | (diverge) |

(Relations 2 and 3 are transposes of each other, and so are 4 and 5.)

However, this did not allow us to simplify the complicated definitions of the conflict and "should be propagated" relations from the original paper [1].

## 2.3 Abstract File Systems and File System Transformers

At the base of our new synchronizer, we envisioned a composable file system abstraction similar to FoxNet [2]. This would make our synchronizer extensible, by allowing us to stack multiple file system *transformers* on top of basic file systems to achieve new functionality. We have thought of basic file systems that:

- operate on a pure finite map data structure;
- operate directly on the local disk.

And file system transformers that:

- make the underlying file system read-only;
- ignore certain paths in the underlying file system;
- mount one file system onto a directory in another;
- save the old version of a file on each edit or remove operation;
- offer a view onto an archive file (Unison or our own);
- offer a view onto a git repository.

As a way to test this abstraction, we envisioned a simple interactive interpreter with commands like `cd`, `mkdir`, `cat`, etc. The goal was to allow this interpreter to run on any supported filesystem, with any meaningful combination of transformers on top.

As a first attempt at implementing this in Haskell, we defined an `FS` type class that would include all monads that implement file system operations. Our goal was to be able to write functions that are polymorphic in the type of file system used. With this approach, however, each call to such a function would have to pass a dummy argument of the form `undefined :: SomeFSType`. Moreover, each of the predicates for the `Ignore` transformer needed to be their own types, since the choice of `FS` was fixed at the type level.

In our second attempt, we made each file system a data value, parametrized by a monadic type. This solved the two problems with our first approach, but left us with no way to thread actions in the monad of the `FS` with actions in `IO`, which would be required for an interactive interpreter.

In our third attempt, we made each file system a monad transformer. This allowed us to apply it to `IO` in the interpreter, and presumably to the identity monad when we did not need interaction with the user. However, this made our file system transformers unsatisfactorily complicated (they now became monad *transformer transformers*). Our main `FS` type class looks like this:

```
class (FSItem i, MyTrans t m, MonadError FSError m)
        => FS fs t m i | fs -> t, fs -> i where
```

where `FSItem` is the type of item stored in the file system, `MyTrans` is a stronger version of `MonadTrans`, and `FSError` is a custom error type defining all errors that can possibly occur in a file system.

Additionally, we have two type classes holding all file systems that work on a pure data structure, and all file systems that have side effects, respectively:

```
class (FS fs t m i) => PureFS fs t m i s | fs -> t, fs -> s where
    run :: fs -> t m a -> s -> m (a, s)

class (MonadIO m, FS fs t m i) => IOFS fs t m i | fs -> t where
    runIO :: fs -> t m a -> m a
```

The `run` function in `PureFS` escapes from the `t m` monad, therefore it needs to provide the state of the `FS`, of type `s`. For `IOFS`, the state is stored in the world, since the underlying monad `m` in a member of `IOFS` can only be `IO`.

# 3 Future Work

There are many avenues that remained unexplored during the ten-week span of this project. This section lists the theoretical and engineering leads that we think are worthy of further exploration.

It would be interesting to extend the algebra with *assert* operations, to see what laws the new operations satisfy, and whether they make the current improvement transformations ($\sqsubseteq$) into equivalent transformations.

We want our synchronizer to support *ignoring* certain paths. We handle this at the file system transformer level, but perhaps it would be advantageous to include this in the algebra.

Our postconditions make claims about sequences of synchronizations. The algebra has no way to express a *"sync occurred"* event, but this is another direction for future inquiry.

Benjamin Pierce suggested a hypothesis about the reconciliation policies, which we have tentatively called the Monotonicity Theorem. In a multi-replica scenario, it is likely that not all replicas participate in any particular synchronization event. These offline replicas, had they been online, should not invalidate the decisions that the synchronizer made in their absence. Only the liberal policy satisfies this intuitive requirement.

Benjamin Pierce also suggested a modified version of our conservative reconciler postconditions, that might hold for the liberal policy. Instead of comparing the sets of all conflicts, he suggested comparing only the top-level paths at which conflicts occur, i.e. if a conflict occurs at $\pi$, then we discard any conflicts at $\pi/\pi'$.

João Dias suggested a possible way to simplify the file system transformer stack, by using monad transformers agnostic of the underlying monad. If this turns out to work, it would be a welcome simplification to the current transformer mechanism.

Coming up with an efficient storage mechanism for a multi-replica synchronizer does not seem trivial. A good solution would maintain efficiency and avoid accumulating large amounts of history if, for example, a replica is offline for a long time and then takes part in a synchronization.

# 4 Acknowledgements

We are grateful to Benjamin Pierce and Daniel Wagner at the University of Pennsylvania for being welcoming hosts during a week-long visit, and for discussions in which many useful ideas emerged. We also thank João Dias and Benjamin Hescott for participating in our daily discussions at Tufts.

# 5 References

1 *An Algebraic Approach to Synchronization*, by Norman Ramsey and Előd Csirmaz

2 *Signatures for a Network Protocol Stack: A Systems Application of Standard ML* by Edoardo Biagioni et al.